第 001 课 不要再用老方法学习单片机和 ARM

第 1 课 学习单片机前途如何？

我们的第一期是教大家如何将 ARM 开发板当作单片机来用，但在这期视频的第一节，我告诉你们，学习单片机是没有前途的。

话说得非常狠，因为不这样说，没法警醒你们。我说这句话，是冒着生命危险的，因为很多人依靠单片机来生活，淘宝上有一大堆售卖单片机开发板的，像 51、STM32 等。

工资方面

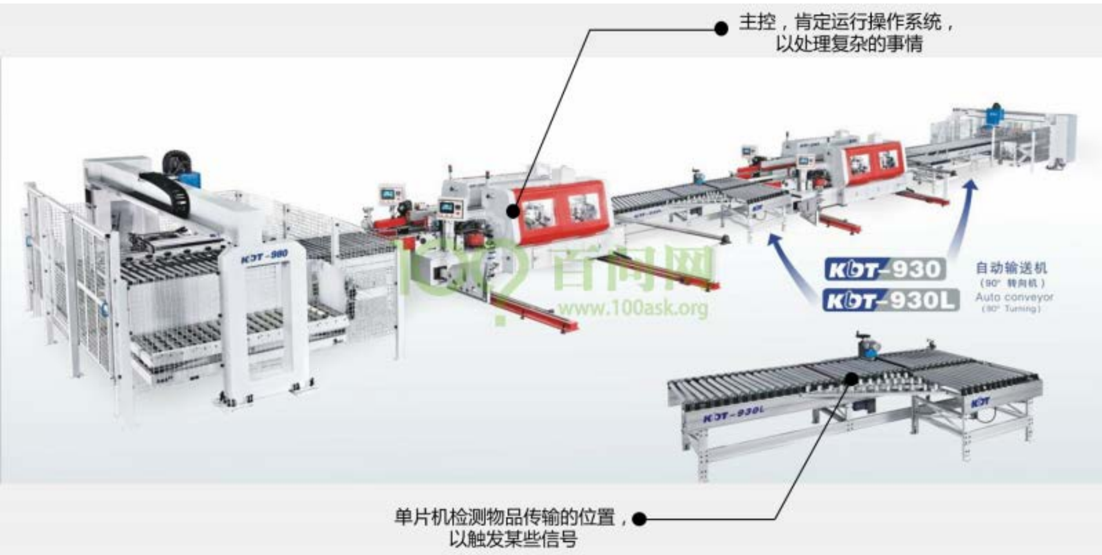
我们学习这些不就是为了赚钱吗？在 51job 上，搜索“单片机”，工资平均下来是几千块钱。然后搜搜“Linux 系统工程师”，平均工资是上万左右。

职业发展

这里我有切实的体会，我 2003 年毕业，2005 年进入一个小公司，当时做的是车载电话，我们先是用 51 单片机来做的。当时我的李姓同事，用两个 3000 多行的 C 文件，实现了车载电话的功能。 另一个魏姓同事，将功能拆分成各种模块，使用了 50 多个 C 文件，以操作系统的思想，重新写了这个程序。这两个牛人，都跳槽了，都不做单片机了。 李姓同事去了美国，深造了机器人视觉，现在是百度的搜索专家，魏姓同事和我一起去了中兴，现在在厦门联想公司负责手机的开发，而我给你们录视频，但我们都不玩单片机了。 我在 2005 发表了一个 2440 开发板上仿照 ucos 写了一个操作系统，10 年前，我们已经把单片机玩得登峰造极了。但是我们还需要升级，为什么？ 因为单片机非常简单，稍微认真学习 2-3 个月就可以达到中等的水平，你工作十年和工作两年技能差别不大，对一个公司，现实一点，他肯定喜欢使用工资更低还更愿意加班的新人，所以说，我们必须升级。

在嵌入式领域，单片机位于哪个位置。我们看看一个自动化的机床，在这条生产线上面，比如说在这传输带上，会有一些单片机来检测物品传输的位置，触发某些信号。 但其总控肯定运行操作系统，以处理更加复杂的事情 。

应用方面



再看看 Google 的机器人，他的手臂、脚趾，也许有些单片机来控制其动作，但他的核心大脑，肯定运行操作系统，才能处理各种复杂的事情。 也就是说，在一个复杂的系统，操作系统是大脑，单片机是可以实现手指的功能。那么你想去做大脑还是想去做手指？



并且单片机的产品升级换代比较慢，一个产品使用单片机的话，他追求的是稳定与成本，那显然，日常的工作的活少，并且也不需要你有太新的技能，既然如此，公司干嘛要花 2~3 万来请人做单片机，直接花几千元请个新人不就完了吗？

并且单片机的价格优势正在逐渐消失，我们在淘宝搜索一下，STM32 开发板的基本都是 1-200 元，而一个能够运行 Linux 系统的板子，nanopi 的价格却在 100 元左右。 一个能够运行 Linux 操作系统，有 512M DDR 内存，有四核处理的的 ARM 开发板，他只需要 99 元。所以说，现在单片机的价格优势正逐渐消失，他只能保持微弱的优势就是稳定性这一特点。

下面来看看使用的单片机和使用操作系统开发的产品，比如闹钟、自动售货机就是单片机做的。但自动售卖机，他一旦需要连接网络，需要 WiFi，他很可能就需要操作系统。现在的新型自动售卖机，上面有微信支付，就必须要操作系统。像无人机，既用单片机也用 Linux 操作系统。单片机可以更加及时的处理一些信息。看这些产品，你更想做哪些产品?



技术方面

我们的操作系统 Linux，他需要一个 Bootloader，这个 Bootloader 就是一个单片机裸板程序的大全，只要掌握了 Linux 的 Bootloader，对单片机是轻而易举。 在后面视频，我会讲解这点。所以说，我告诉你们，学单片机没前途了。

当然我说的是一般情况，你说特例我就完蛋了，周立功做单片机的，年收入几亿，这没办法说。注意，我说的是学习，在学习上，你不需要用单片机来学习，但是在工作中，我们设计产品的时候，如果单片机的性能更好，我们就要选择单片机。 就比如说小米的智能插座，他就是使用单片机来做的，如果同一个功能，用单片机可以省成本，我干嘛不用单片机呢？卖出几百万台设备，每一个省一毛钱，就可以省几十万。我只告诉你，在技术方面，一旦我们掌握了 Linux 的 bootloader,反过来，对单片机来说，他是小菜一碟。

第 2 课 没有前途为何还要学习单片机

为什么没前途也要学习单片机？

因为它是个很好的入口。

单片机的学习可以让我们抛开复杂的软件结构，先掌握硬件操作，如：看原理图、芯片手册、写程序操作寄存器等。 在上一节视频里，我刚把单片机贬得一无是处，说单片机没前途了，这节视频，我又要告诉你们，没有前途，也要学习单片机。为什么？

首先，我说不用学习单片机，是指不要使用老一套得学习方法学习单片机。什么叫老一套的方法？

•硬件上：不要使用 C51、STM32 这些专用的单片机开发板。如果以后，你不打算从事单片机开发，你用这些芯片干嘛，研究了两三个月，把这些寄存器都用清楚了，你又用不上，没必要啊。

•软件上：不要使用 Keil、MDK 等集成度太高的软件。你用这些软件，你写个 main()就可以了，然后调用各种库，进行傻瓜式操作。这些好用的工具，封装了很多技术细节，使得我们没法了解裸机、单片机的本质。

以后我们会使用新一套的方法来进行单片机的开发。新一套的方法，我们后面再介绍。



我们之所以还要学习单片机，是因为他里面的知识，对我们后续学习 Linux 还是有用的。我们首先来看看，一个 Linux 系统是怎么一回事。 一个嵌入式 Linux 系统的软件组成：

单片机大全 Bootloader-->Linux 驱动-->Linux APP-->Linux GUI(Android/QT)

我们 PC 机一上电的时候，黑色屏幕上会显示 BIOS，这个 BIOS 目的是去启动 Windows 内核。Windows 内核再挂载 C 盘(系统盘)、D 盘(应用盘)，最后再去启动应用程序，像 QQ、网游等。

同样的道理，我们的 Android 手机或者工控设备，也有 BIOS，但嵌入式 Linux 系统里面不叫 BIOS，叫 Bootloader，他的目的是去启动 Linux 内核。 他首先也是识别应用程序所在的存储设备，挂载根文件系统（在 Windows 系统里面的 C 盘、D 盘，在 Linux 里面称为根文件系统)。最后去启动应用程序。



仔细的分析下 Bootloader，他去启动内核，他去哪里启动内核呢？ 显然是去某个地方读出内核，就比如说 BIOS 是去 C 盘上读出 Windows 内核，我们的 Bootloader 是去 Flash 或者 SD 卡读取内核。 因此 Bootloader 要拥有读取 Flash

或者 SD 卡的能力。有些 Bootloader 还要显示 logo，因此还要具有操作 LCD 的能力。Bootloader 还要设置开发板的环境，比如，初始化时钟、初始化内存、还要设置网卡等。 这么多事情，都是在 Bootloader 里面实现的，太复杂了，如果你一来就分析整个 Bootloader 是非常困难的。

那我们怎么学习呢？ 把他拆开，写出单独的程序，比如：LED 点灯、时钟、网卡、Flash 都单独写个程序来练习，这些不就是单片机程序吗？所以说，Bootloader 是单片机程序的大全。我们为了更好的学习 Bootloader，我们应该事先一个一个练习硬件，当我们熟悉每个硬件后，再组合起来，就是一个 Bootloader。



我们再来看看 Bootloader 启动内核之后，内核再去挂载根文件系统，意味着内核也要有操作硬件的能力，这就是驱动程序。我们首先来看看一个简单的驱动程序是什么样子。 首先我们的应用程序是调用 open()、read()、write()这些标准的接口去访问硬件。那么就进入驱动程序里面，驱动程序里面有对应的 drive_open()、drive_read()、drive_write()。最后在驱动程序里面，去配置硬件。这里以如果是一个 LED 点灯驱动，那么 drive_open()要把 GPIO 设置为输出引脚，drive_read()，返回 GPIO 状态，driver_write()则写 GPIO，让引脚输出高电平或者低电

平。



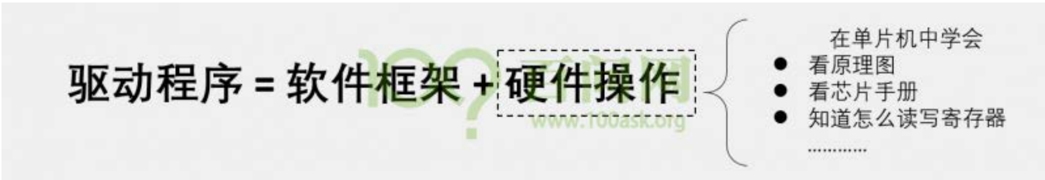
对于我们的 LED 驱动程序，你需要提供 drive_open()、drive_read()、drive_write()这些接口，这就是他的框架。具体的怎么操作硬件，就是硬件操作。 所以说，我们事先在单片机里面，熟悉熟练的掌握硬件操作。即驱动程序的组成：

驱动程序=软件框架+硬件操作

你需要学会看原理图、看硬件怎么连接、看芯片手册、知道怎么读写寄存器。这一切都可以先在单片机里面学习，去掌握。以后学习 Linux 驱动时，把重点放在软件框架就行了。

我们可以事先学习单片机，单片机的学习可以让我们先抛开复杂的软件结构，先掌握硬件的操作，如：看原理图、芯片手册、写程序操作寄存器等。 这就是为什么单片机没有前途，我们也要学习。是因为他里面涉及的硬件操作，对我们后续的学习，非常有用处。

现在我们知道了，我们学习单片机，不是为了掌握单片机的开发技能，而是为了掌握 Bootloader，掌握硬件操作。



第 3 课 怎么学习单片机_选择开发板与开发工具?

本节教你如何学习单片机，如何选择合适的开发板和开发工具。

现在我们知道单片机是要学习的，那么怎么去学习单片机？在上一课我们说不要使用老一套的方法学习，实际上是指的两个问题。

第一：选择什么开发板；

第二：使用什么开发工具；

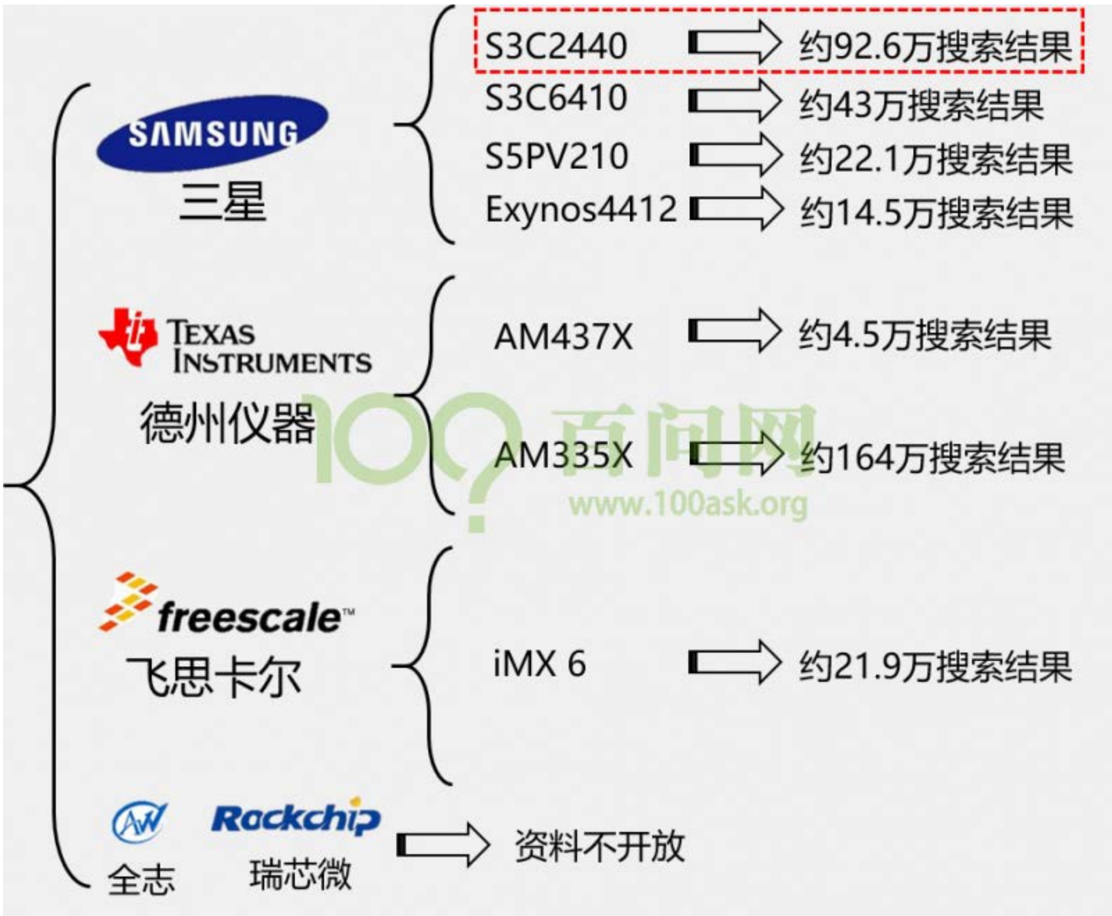
我们学习单片机的目的是干嘛？目的是为后续嵌入式 Linux 学习服务。 在这条学习线路上:

单片机->bootloader->Linux 系统/驱动->APP（QT） 可以使用同一套开发板。

我们选择开发板的原则是：资料丰富。



开发板首推三星 (SAMSUNG)系列的，资料最开放，
有 S3C2440、S3C6410、S5PV210、Exynos4412；
然后是**德州仪器 (TI)**的，TI 开始不开放资料，现在逐渐公开了，有 AM437X、AM335X；
然后是飞思卡尔(freescale)，有 iMX6；
还有其它国产芯片：**全志**、**瑞芯微**；
我们用百度进行搜索，看哪个资料最丰富。结果如下：



再在搜索关键词中加入“教程”后，S3C2440 的搜索结果比 AM335X 更多，因此 S3C2440 的资料仍是最多的，TI 的营销更好，当然，TI 作为工控板，其用户也是很多的。

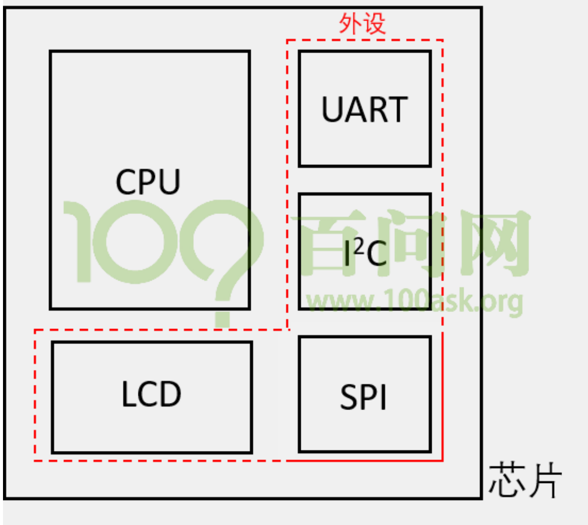
所以，从教程的丰富程度来看，S3C2440 为首选开发板；

我们后面会考虑使用 TI 的芯片重新录制一套视频，但需要 2-3 年的功夫，所以，现在还是使用 S3C2440 吧。

现在普遍有个错误观点：S3C2440 过时了，它还是 ARM9 内核，现在 Cortex-A7、 Cortex-A8、 Cortex-A9、 Cortex-A15 都出来了，它的性能太差了。 性能差没错，但它是否过时了呢？

那要看学到的知识是否过时。

我们首先来看看一个芯片是怎么组成的，里面有 CPU、外设串口、I2C、SPI、LCD 等。



我们写程序的时候，是去操作这些模块的寄存器，访问这些模块，并不是去操作 CPU。 等你工作的时候，你使用不同的芯片，那么差别在于这些模块，这些硬件的操作，差别不在 CPU 操作，你写程序的时候，根本不涉及 CPU 的内部机制。

不同芯片的差别： 在于外设操作，不在于 CPU，写程序几乎不涉及 CPU，只有涉及中断时，才会跟 CPU 的机制有点关系，但是不同 CPU 的架构差异很小，并且我们后面开发 Linux 驱动时，内核已经帮我们做好了这些处理，根本不需要去关心。

从上一课我们知道，驱动=软件框架+硬件操作。这个软件框架对于所有的芯片都是一样的，因为都是用 Linux 内核。 而这个硬件操作，你在 2440 上掌握了串口操作，I2C 操作、SPI 操作，掌握了这些硬件的语言，你换一种芯片，是完全类似的。



我假设你选择了 2440 开发板，那么怎么使用 2440 开发板来学习单片机的开发呢？又使用什么开发工具呢？ 以前在 Windows 开发的时候，我们使用 ADS、Keil、MDK 等，你直接写个 main()函数，所有的细节都帮你实现了，谁来调用 main()函数，有他帮你做了。这 main()所生成出来的代码，怎么放入到内存里面，这工具也帮你做了，我们基本上只需要写 main()函数，只需要写 C 语言就行了。但是这里掩盖了太多的技术细节，你看看我们[官网->学习路线](http://www.100ask.net/a/howtostudy/) 的这篇文章，里面有个比较：

① Windows 下的单片机学习，深度不够

Windows 下有很好的图形界面单片机开发软件，比如 keil、MDK 等。

它们封装了很多技术细节，比如：

你只会从 main 函数开始编写代码，却不知道上电后第 1 条代码是怎么执行的；

你可以编写中断处理函数，但是却不知道它是怎么被调用的；

你不知道程序怎么从 Flash 上被读入内存；

也不知道内存是怎么划分使用的，不知道栈在哪、堆在哪；

当你想裁剪程序降低对 Flash、内存的使用时，你无从下手；

当你新建一个文件时，它被自动加入到工程里，但是其中的机理你完全不懂；

等等等。

② 基于 ARM+Linux 裸机学习，可以学得更深，并且更贴合后续的 Linux 学习。

实际上它就是 Linux 下的单片机学习，只是一切更加原始：所有的代码需要你自己来编写；哪些文件加入工程，需要你自己来管理。

在工作中，我们当然倾向于使用 Windows 下更便利的工具，但是在学习阶段，我们更想学习到程序的本质。

一切从零编写代码、管理代码，可以让我们学习到更多知识：

你需要了解芯片的上电启动过程，知道第 1 条代码如何运行；

你需要掌握怎么把程序从 Flash 上读入内存；

需要理解内存怎么规划使用，比如栈在哪，堆在哪；

需要理解代码重定位；

需要知道中断发生后，软硬件怎么保护现场、跳到中断入口、调用中断程序、恢复现场；

你会知道，main 函数不是我们编写的第 1 个函数；

你会知道，芯片从上电开始，程序是怎么被搬运执行的；

你会知道，函数调用过程中，参数是如何传递的；

你会知道，中断发生时，每一个寄存器的值都要小心对待；

等等等。

你掌握了 ARM+Linux 的裸机开发，再回去看 Windows 下的单片机开发，会惊呼：怎么那么简单！并且你会完全明白这些工具没有向你展示的技术细节。

如果我们基于 ARM+Linux，不使用这些 Windows 工具，你可以学得更深，并且单片机的大全 Bootloader，他就是 ARM+Linux 开发的，他并不使用 Windows 下的工具。 你基于 ARM+Linux 学裸板、学单片机，你可以学得更多，因为我们一切都从零开始的。我们既管理这些代码，也可以知道芯片上电的时候做了什么事情，知道程序自己怎么把自己读到内存，且知道怎么去规划内存，知道怎么代码重定位……

我说的这些概念，你可能听都没听过，这是因为 Windows 下这些好用的工具把这些统统都给屏蔽了。 我们使用 ARM+Linux 进行裸板开发，一旦掌握了 ARM+Linux 开发这套机制，再回过头去看这些 Windows 工具、看 STM32 的话，你只需要几分钟就可以搞定。 并且你可以无缝进入后续的学习，因为你已

经熟练掌握了 Linux 的操作环境，后面的 Bootloader 是在 Linux 下开发的，后面的 Linux 驱动也是在 Linux 下开发的。



所以我们怎么去学习单片机的开发呢？

使用 S3C2440 开发板，在 Linux 环境（Ubuntu）下使用 arm-Linux-gcc 工具来编译程序。

你编写代码的时候，可以使用 Windows 下各种好用的工具，文本工具等，但便宜的时候，使用 arm-Linux-gcc 来编译。

第 002 课 ubuntu 环境搭建和 ubuntu 图形界面操作(免费)

第 001 节_新建目录_新建并编辑文件

首先了解下 Ubuntu 的工具栏，安装好 Ubuntu 进入图形界面后，左边默认有 10 个工具图标，加上我们后面安装的音乐播放器和视频播放器，12 个工具的介绍如下：



Linux 的图形界面操作和 Windows 基本相同。新建文件夹、新建文本文件、编辑、删除等操作几乎都一样的。两者常用操作对比如下：

操作	Windows 系统图形界面	Ubuntu 系统图形界面
新建文件夹	右键菜单	右键菜单
进入文件夹	双击鼠标右键	双击鼠标右键
新建文本文件	右键菜单	右键菜单
打开文本文件	记事本，然后中英文编辑	gedit 编辑器，然后中英文编辑
删除文件	delete 键将文件移动到回收站，shift+delete 键直接删除	delete 键将文件移动到回收站，shift+delete 键直接删除

第 002 节_word_excel_ppt 的操作

Ubuntu 里面也有 Windows 类似的办公软件——LibreOffice。里面包含了Word、Excel、PPT 三件套。



在里面对文档进行编辑和保存，几乎一摸一样，日常使用是几乎没有区别的。

注：Ubuntu 默认是没有安装中文输入法，输入法是我们帮大家安装上的，使用快捷键 Shift+Ctrl 即可切换。

第 003 节_图片浏览_音乐播放_视频播放

图片浏览

在 Ubuntu 里，双击图片文件，默认使用 image Viewer 打开该图片文件。支持常见的 jpg、png、bmp 格式。

音乐播放

Ubuntu 里自带有 Rhythbox 播放器，但对中文支持不太好，这里推荐使用 Audacious。

在 Services->Plugins…->Playlist->Compatibility 中选择 Chinese。

再重新导入音频文件，即可正确显示中文。

视频播放

使用 SMPlayer 打开视频文件，即可观看视频。

第 004 节_网络设置_网页浏览_在线听歌_在线看视频

网络设置

Ubuntu 的网络设置，是初学者不容易掌握的问题，也是百问网近几年答疑频率比较高的问题。

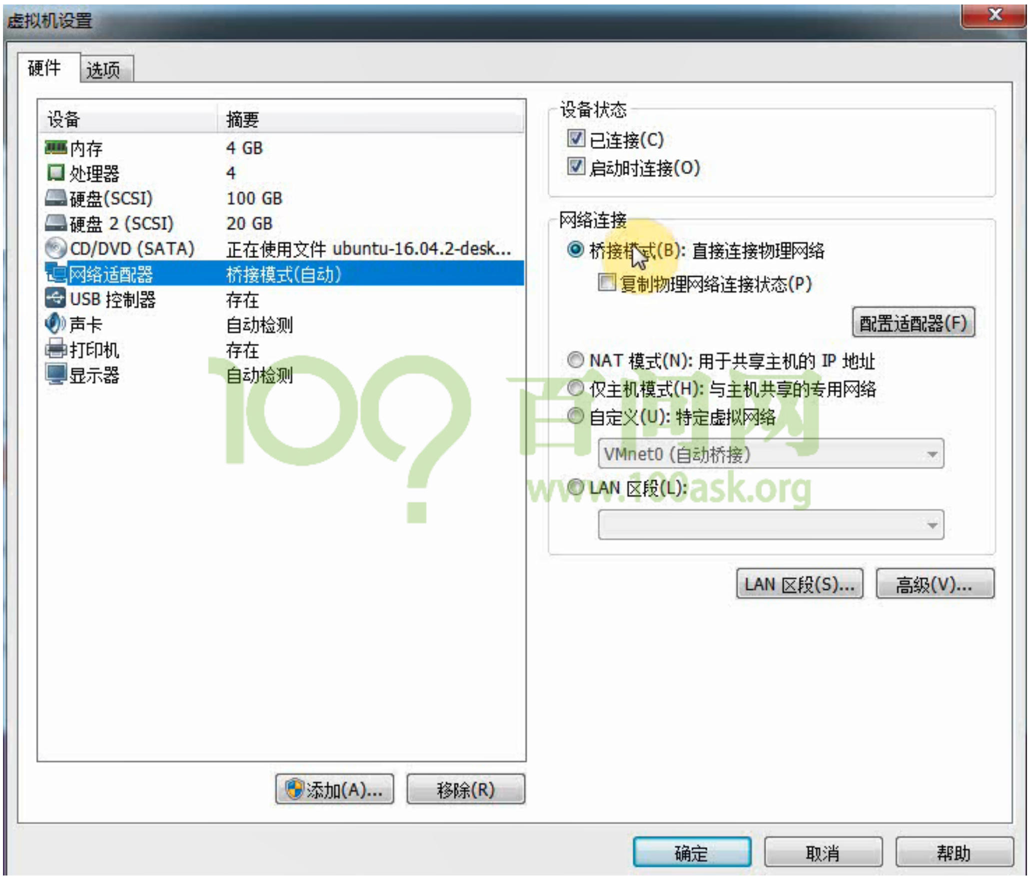
1. 首先，保证 Windows 能够正常上网，例如打开浏览器能够正常浏览网页。

2. 然后设置虚拟机软件：

2.1 打开虚拟机软件，在 Player->管理->虚拟机设置里面，或者使用快捷键“Ctrl”+"D"。

2.2 先选中网络适配器选项，在右边的“设备状态”中勾选上“已连接”和“启动时连接”。在“网络连接”框中，选择“桥接模式”。作为初学者，暂

时可不必理解“桥接模式”的含义，知道这个是虚拟机连接网络最简单的选择即可。



2.3 再点击“配置适配器”按钮，在弹出的窗口里，选中自己电脑实际的网卡。如果不知道自己的网卡，可以右键点击电脑右下角的网络，点击“打开网络和共享中心”，再点击“更改适配器”，可看到自己的“本地连接”，右键本地连接选择属性后，就知道自己连接时使用的网卡了。





3. 设置 Ubuntu：

3.1 点击系统设置图标，选择“Network”，在点击右下角的“Options”。

3.2 点击“IPv4 Settings”选择卡，选择“Automatic(DHCP)”。同时检查虚拟机是否开启了 DHCP 服务：在 Windows 中同时按下“Windows”+“R”键，输入“services.msc”，启动“VMware DHCP Service”。

4. 验证能否联网： 输入 `ifconfig`,查看分配的 IP 地址。这个每个人的可能都不太一样。

先 `ping` 一下内网的路由器：`ping 192.168.1.1 -c 8`

正常的话可以可以看到相关信息。

再 `ping` 一下外网百度：`ping www.baidu.com -c 8`

网页浏览

前面的网络设置好后，就可以通过 Ubuntu 的火狐浏览器，访问外网百度。

网络听歌

网络听歌推荐“网易云音乐”。在浏览器中输入“网易云音乐”，进入首页后，点击下载 Linux 客户端（Ubuntu16.04 64 位），下载保存，再点击下载的安装包输入密码安装即可。 如果安装过程较慢，可以考虑加大虚拟机的内存和处理器数量。

安装完成后，再 Ubuntu 搜索器里面输入“NetEase”即可看到网易云音乐应用程序。之后就可以像 Windows 一样正常使用了。

在线视频

在网页中输入想看的视频，找到视频播放网站，例如腾讯视频。打开视频网页后，提示"Flash 插件已过期，无法播放视频"。这时需要升级 Flash，点击“升级 Flash 插件”即可安装 Flash，然后正常播放视频。

第 005 节_ubuntu 软件下载中心

现在讲解软件中心，打开后，在搜索框输入“audaclous”，可看到该软件，点击软件列表的“Installing”,输入用户密码，就能安装该软件 安装完后，如果再次在列表点击“Removing”,就可以删除该软件。

Ubuntu 软件中心几乎就是“傻瓜式”操作，我相信大家可以掌握。

同理搜索"smplayer",即可下载视频播放软件。

第 006 节_ubuntu 系统设置详细讲解

下面详细的讲解下 Ubuntu 的系统设置。 Ubuntu 的设置分三大块：Personal(个人)、Hardware(硬件)、System(系统)。

Personal 包含：

英文	中文	功能
Appearance	外观	可以设置桌面壁纸等
Brightness&Lock	亮度&锁屏	可以设置锁屏时间
Language Support	语言支持	安装设置语言
Online Accounts	在线账户	后续用到再讲解
Security&Privacy	安全&隐私	后续用到再讲解
Text Entry	文本输入	输入法设置

Hardware 包含：

英文	中文	
Bluetooth	蓝牙	后续用到再讲解
Color	色彩	后续用到再讲解
Displays	显示	可以设置分辨率

Keyboard	键盘	键盘相关
Mouse&Touchpad	鼠标&触摸板	设置鼠标左右键等
Network	网络	前面已经讲解过了网络设置
Power	电源	后续用到再讲解
Printers	打印机	后续用到再讲解
Sound	声音	设置声音大小等
Wacom Tablet	Wacom 手写板	后续用到再讲解

System 包含：

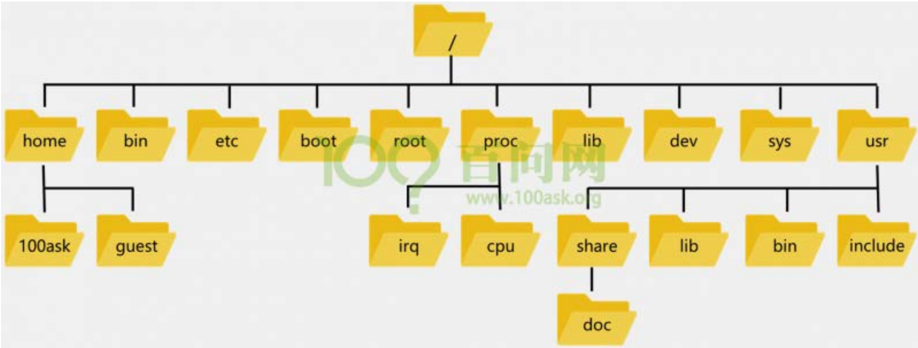
英文	中文	功能
Backups	备份	后续用到再讲解
Details	详细信息	有 Ubuntu 的相关信息
Software&Updates	软件&更新	软件的安装更新
Time&Date	时间&日期	设置系统时间日期等
Universal Access	通用辅助功能	后续用到再讲解
User Accounts	用户账户	后续用到再讲解



第 007 节_Linux 目录结构初体验

打开 Ubuntu 的文件浏览器，默认进入的是 home 目录。 在 Windows 中，有 C 盘，D 盘，E 盘这些概念，需要进入哪个盘，双击进入即可。 在 Ubuntu 下，也是双击文件夹，进入相应的目录。点击 Computer 可以看到很多目录，这些目录在后续再慢慢讲解。

Ubuntu 的树形结构如下：

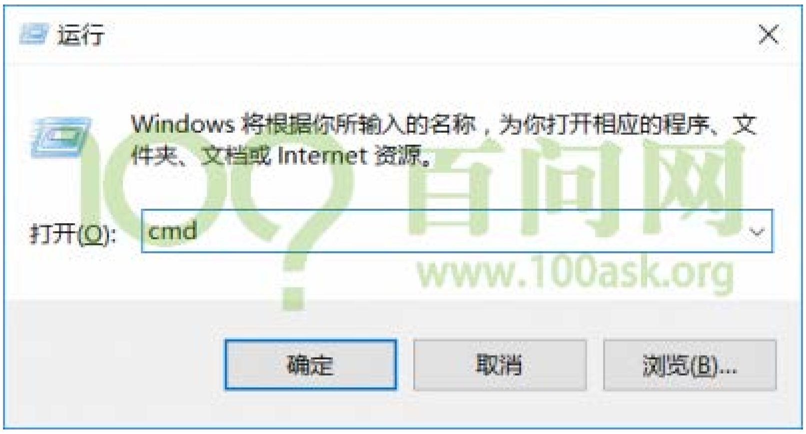


Ubuntu(Centos/Redhat 等其它 Linux 发行版)的目录结构是怎么样的? 都遵循 FHS 标准。

第 003 课 linux 入门命令

001 节_linux 命令入门演示

前面我们讲解了在 Ubuntu 图形界面下怎么新建目录、新建文件等等。
来提个问题：除了这个图形界面，Ubuntu 下面有没有其它的方式也可以创建目录、创建文件呢？
我们回到比较熟悉的 Windows 界面，或许大家听说过 dos 命令行工具，没有听说过也没关系，同时按下“Windows 键”和“R 键”，输入 `cmd`，再点击“确定”，就出现了命令工



具。

打开之后，默认进入计算机用户目录，在该目录输入 `dir`，可以看到显示目录和文件的名称。
执行 `md 123`，在资源管理器可以看到创建了“123”这个文件夹。

执行 `rd 123`,就删除了“123”这个文件夹。

输入 `cls`，即可清除屏幕显示信息。

这里演示的 `dir`、`md`、`rd`、`cls` 这些命令，都是这个命令行所支持的命令。他们可以实现创建目录、删除目录、清屏等操作。

回到前面提的问题：除了图形界面，Ubuntu 下面有没有其它的方式也可以创建目录、创建文件呢？
答案是有的，可以通过 Linux 命令。
下面，我先给大家演示一下 Linux 命令，大家先看着我是如何操作的，在后续视频中我会详细的讲解 Linux 命令如何快速记忆和快速掌握。
现在我们只是来演示一下，在这里强烈建议，不要做笔记，也不要记忆。 演示效果见视频。

在 Ubuntu 下面，我们通过文件搜索器，输入 `terminal`，即可找到终端，或者使用快捷键：Ctrl + Alt + T 也可打开终端。Linux 命令是每一个 Linux 学习者必须掌握的基础知识，入门阶段掌握常用的一些命令即可。 演示中使用到命令：

```
pwd      //显示当前所在的目录
ls       //显示当前目录下的文件
cd       //切换路径
cd ..    //返回上一级路径
mkdir    //新建目录
rmdir    //删除目录
touch    //新建文件
cp       //复制文件
rm       //删除文件
cat      //将文件内容显示到终端中
clear    //清除屏幕的显示内容
```

002 节_shell 命令解析器功能说明

在 Shell 中输入字符串并回车的过程：
1. Shell 根据第一个字符串去环境变量 PATH 所指定的目录中找到同名的应用程

序；
2.然后执行它；
我们可以通过设置 PATH 环境变量来添加我们的目录：

```
export PATH=$PATH:/my/dir
```

command not found 的原因：
1.PATH 所指定的目录中无此程序；
2.它不是可执行的程序；

003 节_linux 命令提示符和 linux 命令格式

这节课我们正式学习 Linux 命令，只有掌握了 Linux 基础命令，我们才能成功迈入 Linux 的世界。
前面演示过一些 Linux 的命令，这些 Linux 命令非常基础，每个同学都必须掌握。我们重新来讲解，并且补充一些 Linux 相关的基础知识。

Linux 的命令提示符

使用快捷键：Ctrl + Alt + T 打开终端，打开终端之后，可以看到它已经显示了一串字符，这一串就是命令提示符。 命令提示符表明了当前终端的状态，这一串的含义如图：



首先 `book` 表示当前登陆的用户名，因为 Linux 是一个多用户的操作系统，所以说指明一下当前是哪一个用户名登陆的。
接下来是一个`@`，它是一个连接符，连接前后。

接着的 `www.100ask.org` 是一个主机名，如果大家没有使用我们百问网提供的 Ubuntu 环境的话，这主机名可能不同，但没有关系。

再接着是一个`:`它也是一个连接符，连接前后。 再接着是一个`~`，它是一个目录，表明了 book 这用户的家目录/home/book，在后面我们会说明。

再接着是一个`$`,因为 book 是一个普通用户，普通用户显示`$`，如果 book 是一个 root（超级）用户，就会显示`#`。这个 root 用户在后面会说明。

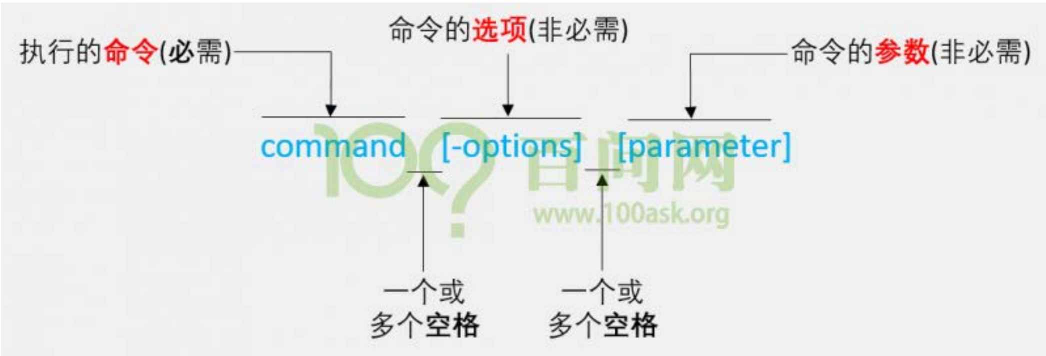
因此我们需要记住两点，命令提示符，显示了：

- 1.当前登陆的用户名
- 2.用户当前所在的路径

Linux 命令的格式

命令　　[选项]　　[参数]

在一些参考书里，是以英文写的，它们是一个意思。



举例： 如果我们只需要查看当前路径的文件内容：`ls`

如果我们想查看当前路径下文件的创建时间和大小等等这些详细内容：`ls -l`

如果我们想查看非当前路径下的内容，比如/home 这个路径下的内容：`ls -l /home`



可以发现：

- 1.命令这一部分肯定是存在的
- 2.选项和参数不一定存在，可有可无；选项和参数取决于里使用该命令实现的具体目的；
- 3.选项是以“-”来指明的；
- 4.命令、选项、参数之间以空格隔开（一个或多个空格都视为是一个空格）；
- 5.完成命令输入后，按下“enter 键”，即可执行命令；

如 `ls` 有个 `-l` 选项，可能有的同学会问，那么 `ls` 到底有那些选项了？

可以通过 `man ls` 来查看。 ls 命令的选项有很多，入门阶段先掌握 ls 命令的-l、-a、-h 等常用选项。 在后面的视频中，也是先介绍 Linux 基础命令（pwd、cd、ls 等）的常用选项。 入门阶段，掌握一些常用的选项就可以了，不要贪多。

004 节_linux 常用命令引入

前面演示过这些命令，这节课开始重新来讲解，在讲解的过程中，我将告诉大家我自己的记忆方法，我相信这些记忆方法，可以帮助大家快速的掌握这些命令。言归正传，这里有几个单词：

序号	英语单词	含义
1	directory	目录
2	change	改变
3	list	列出
4	print	打印
5	remove	删除
6	copy	复制

7	move	移动
8	clear	清除

非常简单，我相信每个同学都能掌握。好了，等下的记忆方法，就是这些单词的组合。

005 节_pwd 命令__独家记忆方法

最简单的是 `pwd` 这个命令，我参考了国内很多讲解 Linux 命令的视频，其实很多老师讲解这个命令的时候，都讲解得不是很完善，他们只是告诉你怎么用，但是他没告诉你怎么来记，它的来源是什么。

我这里要讲解一下它的来源，它是 `print working directory`，这三个单词得首写字母的组合。这三个单词的中文意思就是“打印当前工作路径”。我相信你只要掌握这三个单词的含义，你就能清楚的记住 `pwd` 这个命令，这样的话，你就知根知底了。

在前面讲过，Linux 命令格式是由命令、选项、参数。这三个部分来构成的。在日常工作和学习中，只需要掌握命令部分为 `pwd` 就可以了，至于其它的选项、参数就可以不用掌握了。

演示效果见视频。

```
pwd      //打印当前所在的路径
```

006 节_cd 命令讲解 1__独家记忆方法

接着是 `cd` 命令，这 `cd` 命令的来源是 `change directory` 这两单词的首写字母组合。这两单词的意思就是“切换路径”。一般情况下，`cd` 命令都不需要加参数，只需要加上，需要切换的目录就行了。

演示效果见视频。

```
cd      //切换路径
```

演示过程中，每次都需要输入很多，是不是有时可以简化输入呢？答案是可以的。一般简化输入有这三种情况：

```
cd  ~   //切换到家目录

cd  ..  //切换到上一级目录
```

cd -   *//切换到上一次所在目录*

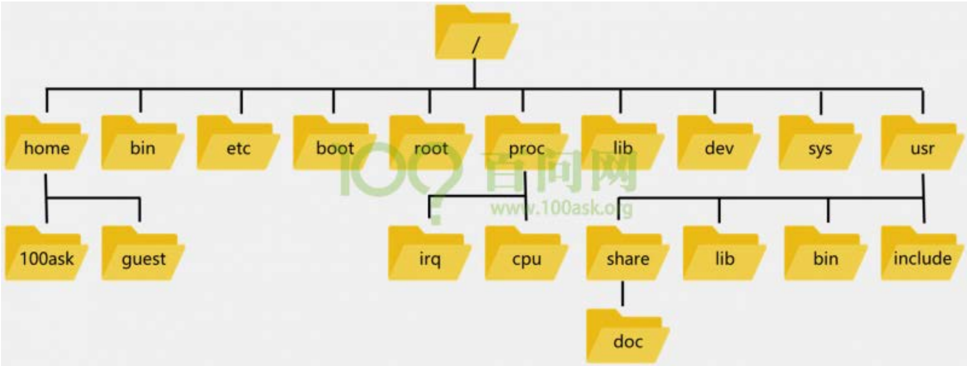
007 节_linux 基础知识__家目录_当前路径

- 第一个概念是**家目录**，比如 book 用户的家目录是/home/book。

这是前面讲解过的 Ubuntu 结构，当你在 Ubuntu 下面新建一个 book 用户后，它会在/home 目录下新建一个 book 目录。

/home/book 这个目录来存放 book 用户他自己的一些文件。

同理，你在 Ubuntu 下面新建 guest 这个用户在话，他也会在/home 下面 guest 这个目录。/home/guest 来存放 guest 用户，他自己的一些文件。



Linux 中用`~`符号来表示用户的家目录。因此在 Linux 中输入 `cd ~`即可进入家目录。

演示效果见视频。

下面都是一些 Linux 的基础概念，我们先补充一下。

008 节_linux 基础知识__上一个路径_上一次路径

- 第二个概念是当前路径和上一级路径。

cd   *//切换路径*

cd ~   *//切换到当前用户的家目录*

cd .   *//切换到当前路径，很少使用*

cd ..   *//切换到上一级路径*

cd ../..   *//切换到上上级路径(其中：在 linux 中路径分隔符为斜杠/)*

cd -   *//切换到上一次的路径*

Linux 中用`.`符号来表示当前目录，用`..`符号来表示上一级目录。

009 节_linux 基础知识__linux 目录结构介绍

Linux 目录结构 参考前面

```
/                ------------------根目录/
|------bin       ------------------/bin
|------home      ------------------/home
   |------book   ------------------/home/book
   |------Guest  ------------------/home/Guest
|------usr       ------------------/usr
   |------lib    ------------------/usr/lib
   |------bin    ------------------/usr/bin
```

010 节_linux 基础知识__绝对路径和相对路径

- 第三个概念是绝对路径和相对路径

绝对路径：从根目录一级级找下去，需要写完整路径名
相对路径：参照当前所在目录进行查找
举例：

1）当前路径为/bin，然后`./pwd`就是执行了`/bin/pwd`

2）当前路径为/home/book/100ask，然后 `cd ../Videos/`就到达

/home/book/Videos，`cd ../..`就到达/home

绝对路径是从根目录开始； 相对路径一般以`.`和`..`来构成；

演示效果见视频。

011 节_cd 命令讲解 2_注意事项

注意事项：

- 1.切换的路径必须保证存在，可以使用`</ls`命令来查看某一个路径下的所有内容。
- 2.一定要记得多使用 tab 键，让终端为你自动补全。

你切换的路径必须保证存在，如果路径都不存在，肯定切换不了。这相当于什么呢。 你们班上有 30 个学生，学号是 1-30 号，老师叫了 31 号，肯定没人应答嘛。

因此，建议切换路径前，使用`</ls`命令来先查看下有没有这个路径，

`</ls`命令，后面会讲。

tab 键可以减少我们的输入，提高我们的效率，我见过很多初学者，每次输入的时候都输入完整的文件名或者路径名，效率很低下。所以，记得多使用 tab 键。

演示效果见视频。

012 节_ls 命令__独家记忆方法

接下来，我们讲解`ls`命令。 `ls`命令它的来源是英文单词`list`。学过数据结构的同学知道，`list`表示链表，不过在这里，`list`是个动词，表示列举、列出。`ls`的功能是列出目录内容。

```
ls    //列出目录内容
```

前面我们刚刚讲过`cd`命令，`cd`命令是切换路径。 一般的话，都是用`cd`切换路径，然后再使用`ls`来查看目录内容。

在前面讲过，Linux 的命令组成是：

```
命令    [选项]    [参数]
```

其中选项，参数可有可无。

举例：

1） ls

```
ls          //则显示当前所在目录的内容
```

2)ls　目录名

```
ls /home//则指定显示/home 的内容
```

ls ~　　*//则指定显示/home/book 的内容*

3)ls　选项　或　ls　选项　目录名 常用的选项：

`-l`(long 的缩写)，显示目录下文件的更详细的信息(文件权限、文件最后修改时间、文件大小)；

`-a`(all 的缩写)，显示了隐藏文件；

`-h`(human-able 的缩写)，将文件大小以 K(KB)、M(MB)、G(GB)来表示；

`ls -lh` 的详细的信息含义：



演示效果见视频。

013 节_目录操作 mkdir 和 rmdir_独家记忆方法

下面我们讲解 mkdir 这个命令。`mkdir` 来源于 `make directory` 目录的操作

- mkdir：创建目录

举例：
1)mkdir 目录名

mkdir dir0　　*//创建了 dir0 这个目录*

2)mkdir -p 父目录/子目录

mkdir -p dir1/dir2 *//连续创建多级目录(父目录和子目录)，如果父目录不存在，则需要加入-p 参数。-p(parents 的缩写)*

- rmdir ：删除目录

举例：
1） rmdir 目录名

```
rmdir dir0    //删除dir0 这个没有内容的空目录
```

注意：rmdir 不能删除非空目录(非空目录:该目录下面有子目录或者文件)
2）rm -f 目录名

```
rm -f dir0   //删除dir0 这个目录
```

注意：可以删除非空目录
演示效果见视频。

014 节_文件操作 touch_mv_cp_rm

文件的操作

- touch :用来新建文件

举例：
1） touch 文件名

```
touch file1    //在当前目录新建名为file1 的文件
```

注意：
1)同一目录无法创建同名的文件
2)linux 的文件名是区分大小写的，如 file1 和 File1 是不同文件。这一点跟 windows 不同。

- mv(move 的缩写) :用来修改文件(目录)名、移动路径

举例：
1） mv 旧文件名 新文件名 修改文件名

```
mv file1 filea  //将当前目录的文件 file1 改名为 filea
```

2）mv 旧目录名 新目录名 修改目录名

mv dir1 dira *//将当前目录下的目录dir1 改为dira*

3）mv 文件名 目录名 移动路径

mv filea dira *//将当前目录下filea 移动到dira 子目录*

mv filea ~ *//将当前目录下filea 移动家目录*

mv ~/filea .. *//将家目录下的fila 移动到上一个路径*

- cp(copy 的缩写) :用来复制文件(目录)

1）cp 源文件名 目标文件名

cp File1 file2 *//将当前目录下的文件File1 拷贝成file2*

2）cp 源文件名 目标目录名

cp file2 dira/ *//将当前目录下的文件file2 拷贝到dira 子目录*

2）cp 源目录名 目标目录名

cp -r dira dirb *//复制dira 目录下的所有内容到dirb,其中,-r 参数时递归复制*

cp -i file2 dira *//如果dira 目录下面有同名的file2，加入-i 参数就会要求你确认是否覆盖同名的文件*

注意：cp 的常用参数有：-i,-r,-f,-d 等，在后面讲解了 linux 权限之后，我们会加深 cp 这个命令的讲解。

- rm(remove 的缩写) :删除文件(目录)

常用命令格式：

rm [选项] [文件名 | 目录名]

常用选项： `-i`(interactive(交互)的缩写),删除文件(目录)之前，要求你确认是否同意删除 `-r`(recursive(递归)的缩写),递归删除指定目录下的子目录和文件 `-f`(force(强制)的缩写),强制删除

举例：

```
rm File1      //删除当前目录下的文件File1
rm -i file2   //删除文件file2，删除前要求你确认是否同意删除，其中y表明同意删除，n表示取消删除
rm -r dira    //删除目录dira
rm -ir dirb   //删除目录dirb，删除前要求你确认是否同意删除，其中y表明同意删除，n表示取消删除
```

注意：

1）删除文件(目录)前，确定该文件(目录)是否可以被删除。

2）rm的常用参数有：-i,-r,-f等，在后面讲解了linux权限之后，我们会加深rm这个命令的讲解。

015节_文件查看和编辑 cat_gedit

文件的查看和编辑：

1） cat用来查看文件内容

常用命令格式：

```
cat  文件名
```

举例：

`cat file1`　　将file1的内容打印到标准输出中(默认标准标准输出指向终端)

`cat file1 file2`　　将file1和file2的内容串联并依次全部打印到标准输出中

`cat -n file1`　　显示内容并在内容前显示行号。

类似的查看命令:more、less、head、tail 等后面讲解

2) gedit 图形应用程序的编辑器
这个前面讲过了，就不再重复了

3) vi 编辑器
vi 编辑器非常重要，在后面的视频，以专题的形式讲解。

016 节_清除屏幕_clear 和 reset

清除屏幕命令：

`clear`：刷新屏幕，保留历史命令操作记录

说明：此命令本质上只是让终端向后翻一页，当向上滚动鼠标时，还是可以看到之前命令的操作记录 也可使用快捷键“Ctrl”+“L”。

`reset`：重新初始化屏幕，清除历史命令操作记录

017 节_帮助信息 man_info_help

帮助信息：
学习 Linux 最麻烦的是什么?
不停的上网或者翻阅书籍查找 linux 命令、相关函数的用法。
其实，没那么麻烦，linux 发行版(如 ubuntu)自带帮助命令。
Linux 系统中提供了三种帮助方式。
man 和 info 是独立的命令， --help 是个命令的参数， 它们都是 Linux 中获取帮助信息最权威，最快捷的途径。

1) man 使用的最多
举例：
man man　　//查看 man 手册的说明
man ls　　//当没有指定使用那一页，默认使用第 1 页
man 1 ls　　//与 man ls 一样
man 1 gcc　//gcc 是一个应用程序，在 linux 中一般使用 gcc 编译器来编译 c/c++语言的程序
man 2 open　//查看系统调用 open 的 man 手册说明。open/write/read/close 等等都是系统调用
注意： man 手册的 9 册内容的侧重点，最好记一下。

section	名称	说明

1	可执行程序或 shell 命令	用户可操作的命令
2	系统调用	内核提供的函数(查头文件)
3	库调用	常用的函数库
4	特殊文件	在/dev 下的设备文件
5	文件格式和约定	对一些文件进行解释，如/etc/passpd
6	游戏程序	游戏程序
7	杂项	包括宏包和约定等
8	系统管理员使用的管理命令	通常只有系统管理员 root 可以使用
9	内核相关	Linux 内核相关文件

2) info
举例：
info ls //查看 ls 的帮助信息
3) --help
举例：
ls --help //查看 ls 的帮助信息

第 004 课 vi 编辑器

vi 编辑器 1：一个编辑器具备的功能
一个编辑器(例如 Windows 中的记事本)具备的功能：
打开文件、新建文件、保存文件
光标移动
文本编辑
(多行间|多列间)复制、粘贴、删除
查找和替换
vi 编辑器 2：vi 编辑器的环境设置
为了更方便的使用 vi 编辑器，我们需要先对 vi 编辑器进行一些配置。打开虚拟机终端，输入以下命令：

```
cd /etc/vim          //进入 vi 配置文件目录
cp vimrc ~/.vimrc    //复制配置文件到我们自己的家目录(这样配置文件只会对我们自己有效，不会影响其它用户)
cd ~                 //进入到家目录
gedit .vimrc         //使用 gedit 编辑器编辑.vimrc 配置文件
```

在.vimrc 中加入如下内容：

"关闭兼容功能

set nocompatible

"显示行号

set number

"编辑时 backspace 键设置为 2 个空格

set backspace=2

"编辑时 tab 键设置为 4 个空格

set tabstop=4

"设置自动对齐为 4 个空格

set shiftwidth=4

"搜索时不区分大小写

set ignorecase

"搜索时高亮显示

set hlsearch

保存，退出。

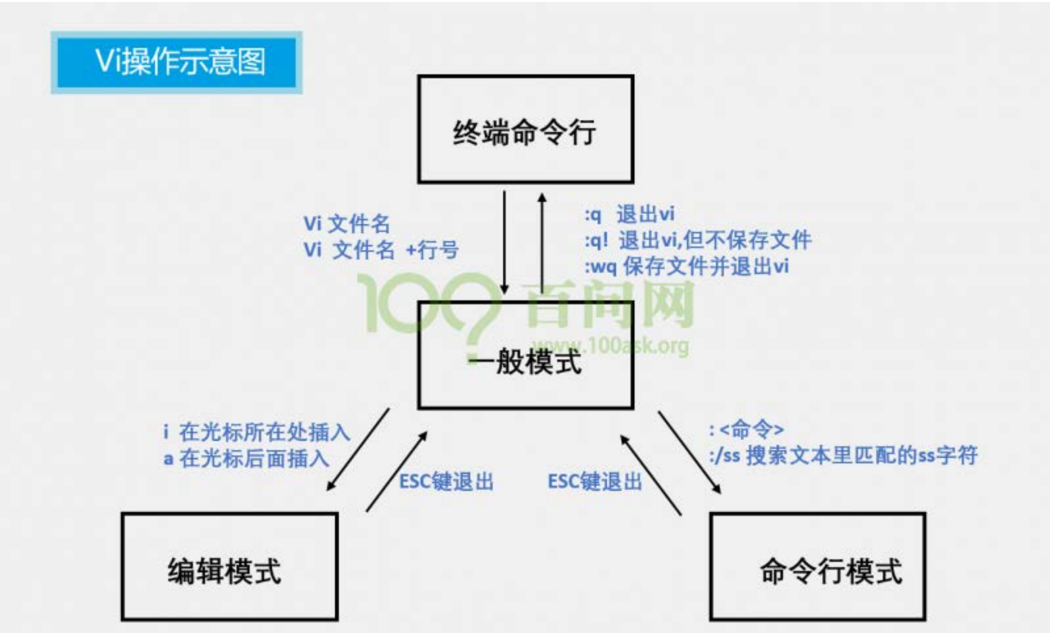
vi 编辑器 3：vi 编辑器的三种模式__模式间相互切换

vi 编辑器有三种模式,各个模式侧重点不一样。

一般模式（光标移动、复制、粘贴、删除）

编辑模式（编辑文本）

命令行模式（查找和替换）



操作演示见视频。

注意：
当不知道处于何种模式时，按 ESC 键返回到一般模式。
ccwq(write quit)
i(insert)
vi 编辑器 4：文件的打开_新建_保存
1.打开文件、新建文件、保存文件

`vi 文件名`

如果文件存在，输入结束后，`:wq`保存并退出文件

如果文件不存在，输入结束后，`:wq`就可以新建并保存文件

在编辑完成时，返回一般模式，

输入`:w`则保存文件，如果已经保存文件，输入`:q`则退出文件

直接输入`:wq`保存并退出

如果不想保存被修改的内容，则`:q!`强制退出。

2.进入编辑模式
在一般模式输入：

`i`(在光标前开始插入文本)

`a`(在光标后开始插入文本)

`o`(在当前行之下新开一行，并到行首)

vi 编辑器 5：如何使 vi 快速移动光标__vi 的难点
3.光标移动 在一般模式下，hjkl 这四个按键就可以移动光标
h (左)
j (下)
k (上)
l (右)
1）快速的**定位到某一行**：
文件头、文件尾、指定某一行
ngg　　//光标移至第 n 行的行首（n 为数字,想要跳转的行），
1gg　　//就跳到第一行的行首，就是文件头
2gg　　//就跳到第二行的行首
G　　　//转至文件结尾
注意：ngg 和 G 是在一般模式
2）在某一行如何快速**定位到某一列**：
0　　//（数字零）光标移至当前行行首
$　　//光标移至当前行行末

fx　//搜索当前行中下一个出现字母 x 的地方

注意：0、$、fx 是在一般模式

vi 的难点：

vi 操作之前，先判断一下当前是哪一种模式，再看光标所在位置。 当你不知道处于何种模式时，使用 esc 键返回到一般模式。 再看光标，难点在于移动光标，可以做到快速切换到某一行某一列。

vi 编辑器 6：文本复制_粘贴_删除_撤销__百问网独家记忆方法

4.文本复制、粘贴、删除、撤销

复制

yy　　//复制当前行(y:yank(复制))

nyy　　//复制当前行及其后的 n*1 行(n 是数字)

粘贴

p　　　//粘贴(p:paste)

删除

dd　　//删除光标所在行(d:delete)

ndd　　//删除当前行及其后的 n*1 行(n 是数字)

x　　　//删除光标所在位置的字符

撤销

u　　　//撤销上一步操作

注意：

yy/nyy/dd/ndd/x/u 都是在一般模式。

操作演示见视频。

vi 编辑器 7：文本查找和替换__百问网独家记忆方法

5.查找和替换

查找

/pattern　　//从光标开始处向文件尾搜索 pattern，后按下 n 或 N

注意：

n 在同一个方向重复上一次搜索命令

N 在反方向重复上一次搜索命令

n(N)来源于 next

注意： 在/pattern 之前先跳到第一行则进行全文件搜索。

替换

:%s/p1/p2/g　　//将文件中所有的 p1 均用 p2 替换

:%s/p1/p2/gc　　//替换时需要确认

s :substitute 替换

g :global 全局

c :confirm 确认

操作演示见视频。

vi 编辑器 8：vi 难点回顾__实例演示 2

vi 的难点：

1）vi 操作之前，先判断一下当前是哪一种模式，再看光标所在位置。 2) 当你不知道处于何种模式时，使用 esc 键返回到一般模式。 3) 再看光标，难点在于移动光标，可以做到快速切换到某一行某一列。

vi 编辑器有三种模式及各模式常用操作

1）一般模式（光标移动、复制、粘贴、删除、撤销）

hjkl、ngg/G、0、$、fx

yy/nyy
p

dd/ndd、x

u
2）编辑模式（编辑文本）
i　a o
3）命令行模式（查找和替换、保存退出文件）

/pattern、:%s/p1/p2/gc、:wq

举例：
用 vi 新建一个文件 test2.txt，然后输入”welaome to the world of vi.”,并保存。
由于拼写错误，需要将其中的 welaome 中的 a 替换成 c。
在上步的基础上，复制第一行到第二行和第三行。
在上步的基础上，删除第三行的全部内容。
在上步的基础上，搜索出现 vi 的地方。
在上步的基础上，将字符串 vi 替换成 vim。
保存并退出。
列出压缩文件的内容
-k(keep)　　在压缩或解压时，保留输入文件。
-d(decompress) 将压缩文件进行解压缩
1）查看
gzip -l 压缩文件名
比如： gzip -l pwd.1.gz

2）解压
gzip -kd 压缩文件名
比如： gzip -kd pwd.1.gz 该压缩文件是以.gz 结尾的单个文件

3）压缩
gzip -k 源文件名
比如：gzip -k mypwd.1 得到了一个.gz 结尾的压缩文件

注意：
1）如果 gzip 不加任何选项，此时为压缩，压缩完该文件会生成后缀为.gz 的压缩文件，　并删除原有的文件，所以说，推荐使用 gzip -k 来压缩源文件。
2）相同的文件内容，如果文件名不同，压缩后的大小也不同。
3）gzip 只能压缩单个文件，不能压缩目录。

提示：

man pwd 会解压/usr/share/man/man1/pwd.1.gz 这个文件， 然后读取该文件中固定的格式的一些信息，然后显示到终端中。

bzip2 来压缩单个文件
bzip2 的常用选项:
-k(keep) 在压缩或解压时，保留输入文件。
-d(decompress) 将压缩文件进行解压缩
1）压缩
bzip2 -k 源文件名
比如：bzip2 -k mypwd.1 得到一个.bz2 后缀的压缩文件

2）解压
bzip2 -kd 压缩文件名
比如：bzip2 -kd mypwd.1.bz2

注意：
1）如果 bzip2 不加任何选项，此时为压缩，压缩完该文件会生成后缀为.bz2 的压缩文件， 并删除原有的文件，所以说，推荐使用 bzip2 -k 来压缩源文件。
2）bzip2 只能压缩单个文件，不能压缩目录。

单个文件的压缩使用 gzip 或 bzip2
压缩有两个参数：
1）压缩时间
2）压缩比

一般情况下，小文件使用 gzip 来压缩，大文件使用 bzip2 来压缩。

比如：
mypwd.1 源大小是 1477 字节，
gzip 压缩后 mypwd.1.gz 是 877 字节，
bzip2 压缩后 mypwd.1.bz2 是 939 字节。

myls.1 源文件大小 7664 字节，
gzip 压缩后 myls.1.gz 是 3144 字节，
bzip2 压缩后 myls.1.bz2 是 3070 字节。

gzip、bizp2 只能对一个文件进行压缩，而不能对多个文件和目录进行压缩。所以需要 tar 来对多个目录、文件进行打包和压缩。

tar 常用选项
-c(create) 表示创建用来生成文件包
-x：表示提取，从文件包中提取文件

-t 可以查看压缩的文件。 -z 使用 gzip 方式进行处理，它与”c“结合就表示压缩，与”x“结合就表示解压缩。

-j 使用 bzip2 方式进行处理，它与”c“结合就表示压缩，与”x“结合就表示解压缩。

-v(verbose)详细报告 tar 处理的信息

-f(file)表示文件，后面接着一个文件名。

-C <指定目录> 解压到指定目录

1.tar 打包、gzip 压缩

1）压缩

tar -czvf 压缩文件名 目录名

如： tar czvf dira.tar.gz dira

注意：tar -czvf 与 tar czvf 是一样的效果，所以说，后面统一取消-。

2）查看

tar tvf 压缩文件名

如：tar tvf dira.tar.gz

3）解压

tar xzvf 压缩文件名

tar xzvf 压缩文件名 -C 指定目录

如： tar xzvf dira.tar.gz 解压到当前目录

如： tar xzvf dira.tar.gz -C /home/book 解压到/home/book

2.tar 打包、bzip2 压缩

1）压缩

tar cjvf 压缩文件名 目录名

如： tar cjvf dira.tar.bz2 dira

2）查看

tar tvf 压缩文件名

如： tar tvf dira.tar.bz2

3）解压

tar xjvf 压缩文件名

tar xjvf 压缩文件名 -C 指定目录

如： tar xjvf dira.tar.bz2 解压到当前目录

如： tar xjvf dira.tar.bz2 -C /home/book 解压到/home/book

第 006 课 开发板熟悉与体验(免费)

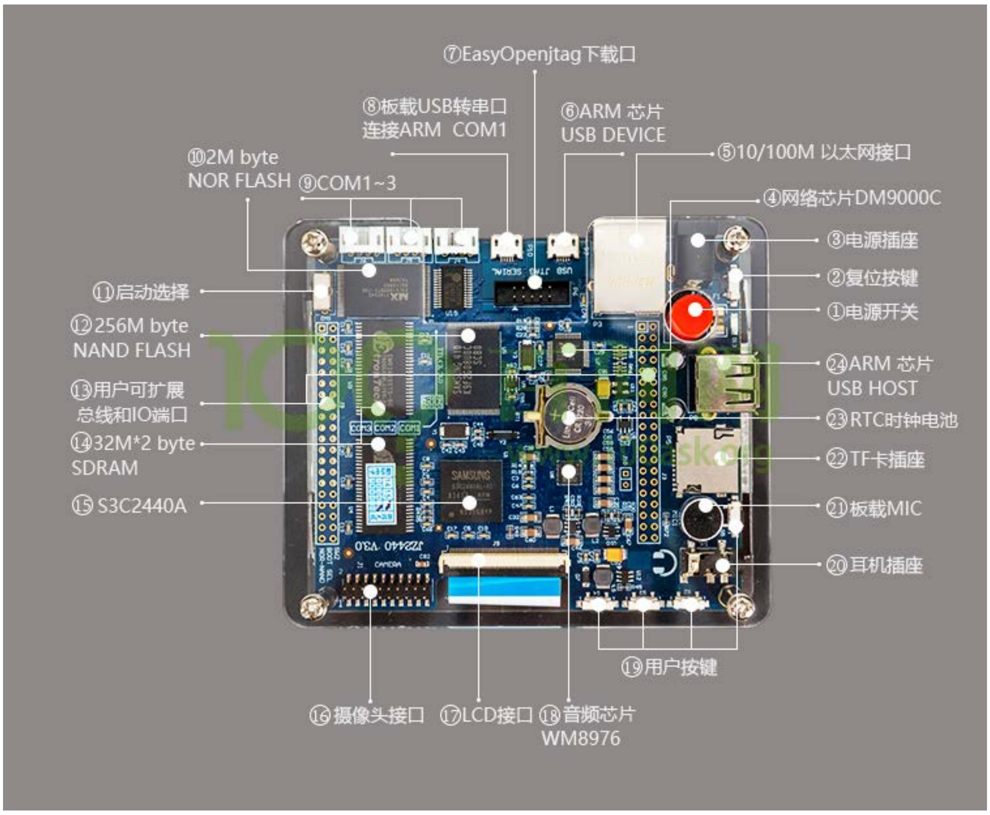
第 001 节_开发板接口介绍与串口连接

在前面的视频里，我们涉及四个接口，两个开关。

四个接口：电源接口、USB 串口、USB 下载口、JTAG 下载口；

两个开关：电源开关、启动选择开关；

分别对应下图中的 1、8、6、7、1、11；



我们买开发板的目的就是把电脑上编写编译好的程序烧写到板子上验证学习。因此开发板上一定有个烧写口，例如 JTAG 烧写口。但电脑上是不会有这个 JTAG 口的，因此需要一个 USB 烧写器将两者连接，例如 Jlink、OP/EOP。Jlink 本来用的人很多，但随着版权意识的提高以及 Jlink 公司对盗版的打击，Jlink 现在用得越来越少了。EesyOpenJtag 是 OpenJtag 的便宜版本，他和我们的开发板是绝配，他可以直接烧写 Nand Flash 和 Nor Flash，操作简单，价格便宜。

对于程序员来说，我们程序都是三分写，七分调，调试非常重要。对于我们电脑，我们可以在屏幕上显示，对于我们开发板，我们可能连显示屏都没有，并且屏幕的操作比较复杂，不太适合用于调试。那我们这里就需要一个比较简单的设备提供调试信息——串口。通过串口，可以把 PC 上的数据传到开发板，开发板也可返回数据。开发板的串口，不能直接与电脑的 USB 相连，中间需要一个串口转 USB 的芯片，这个芯片集成到了开发板上，因此可以直接使用 USB 线将电脑与开发板连接进行通信。

现在再说一下烧写，我们之前说可以通过 Jtag 进行烧写，它非常的可靠，当我们板子变成了砖头的时候，可以使用 Jtag 进行烧写，但他的速度很慢。当我们烧写很大程序的时候，会崩溃的。因此我们可以借助 USB 口进行烧写，前提是板子上已经运行有一个程序，这个程序用来支持 USB 下载

接电源，按下开关；

使用串口（USB 串口）观察信息；

使用 JTAG（USB 烧写器）烧写程序；

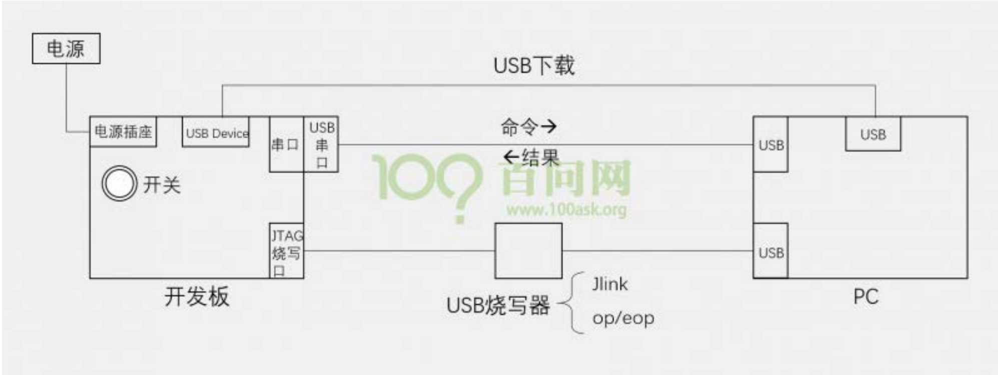
如果板上程序支持 USB 下载，可以使用板子的 USB Device 连接电脑下载；

启动选择开关；

对开发板上电启动，出厂的时候默认烧写一个 Linux 系统。我们上电后可以看到 Linux 的企鹅图标，然后自动启动 Qt。

再将开发板的串口与电脑相连，自动/手动安装好串口驱动后，使用 MobaXterm 软件的 Serial 功能，即可通过串口 输入 Linux 命令。

连接示意图如下：



第 002 节_使用 eop 烧写裸板程序

烧写裸板程序，需要用到的软件是 oflash.exe，使用到的硬件是 op/eop(easy open Jtag)，以及相应的驱动程序。 操作步骤如下：

1. eop 连接到 PC；
2. 安装驱动；
3. 安装 APP；
4. 开发板的排线连接到 eop，eop 的 USB 接口连接到电脑；
5. 执行：oflash xxx.bin
6. 开发板断开 eop,eop 的 USB 接口最好也断开电脑；
7. 设置从 Nor/Nand 启动；
8. 重新上电；

第 003 节_eop 常见问题

eop 常见问题通常有如下 7 个，我们可以根据 oflash 的提示信息，来大致判断是何种情况导致的问题。

1. 未连接 op/eop 到电脑；
2. 有其他程序在使用 op/eop (同一时间只能有一个程序使用它)；
3. jtag 线未接；
4. 开发板未上电；
5. oflash xxx.bin 时当前目录下没有 xxx.bin；
6. 烧写完后没有正确设置启动开关；
7. 烧写完后，op/eop 与开发板之间的排线未断开, 导致程序无法运行；

- 当 oflash 显示 `unable to open ftdi devive:2`,

则可能是未连接op/eop到电脑和有其他程序在使用op/eop。

- 当oflash显示`Description: USB<=>JTAG&RS232 ANo CPU detectred, cupID = 0xffffffff`,

则可能是Jtag线未接。

- 当oflash显示`Description: USB<=>JTAG&RS232 ANo CPU detectred, cupID = 0x00000000`,

则可能是开发板未上电。

- 当oflash显示`ERROR: can't find the file : xx.bin.`,

则可能是当前路径下没用xx.bin，应该使用绝对路径或者复制到指定目录再切换到该目录。

- 当烧写好程序后，开发板上电后却没有运行,

则可能是没有正确设置启动开关或者是op/eop与开发板之间的排线未断开，导致程序无法运行。

第004节_使用uboot烧写裸板程序

在前面，使用eop烧写一个200多k的uboot时，耗费了几分钟，这速度实在太慢了，在后续的开发过程中，我们的程序可能有许多错误，需要反复修改烧写，如果继续用eop将会浪费很多时间。那么有没有更快的烧写方式呢？有的，我们用uboot进行烧写。

在2440上面，有两种flash:Nand Flash和Nor Flash。我们可以首先把uboot烧到Nor Flash，然后Nor启动运行uboot，使用uboot的USB下载功能接收PC传来的文件，然后uboot将收到的文件烧写到Nand Flash，最后Nand启动，启动我们烧写的程序。

操作步骤如下：

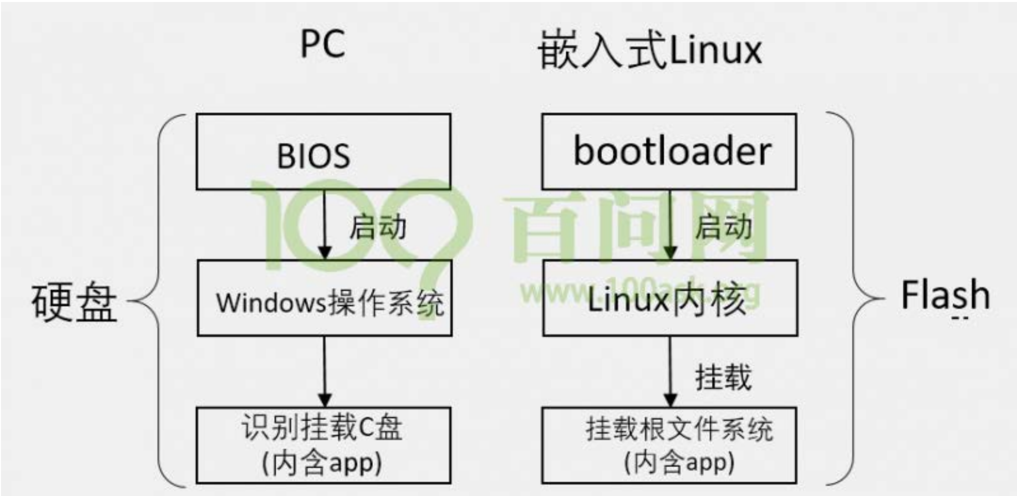
1. 使用op/eop把u-boot.bin烧到nor flash;
2. 开发板设置为nor启动，上电后马上在串口输入空格键，使板子进入UBOOT而不是启动板子上的内核；
3. 连接PC与开发板的usb device口，如果没有驱动，安装驱动；
4. 在UBOOT的串口菜单中输入n (表示接收USB文件并烧写到NAND);

5. 使用 dnw_100ask.exe 发送 bin 文件；
6. uboot 即会自动接收、烧写 bin 文件；
7. 断电、设为 NAND 启动、上电：运行 nand 上烧好的程序；

第 005 节_恢复出厂系统

开发板买来就是学习的，就是用来“破坏的”，不要担心上面的东西被破坏，因为我们有办法恢复出厂系统。

我们先对比 PC 看看出厂系统有哪些东西：



可以看到我们的东西都放在 Flash 上面，对于我们的 JZ2440，有 256M 的 Nand Flash 和 2M 的 Nor Flash，所以我们内核、根文件系统那么多的文件，应该是放在 Nand Falsh。Nand Falsh 内部数据分布如下：



其中 bootloader 既可以在 Nand Flash 也可以在 Nor Flash，params 的变量存储有 uboot 的参数信息。

恢复出厂系统的具体步骤如下：

1. 使用 op/eop 烧写 u-boot 到 nor/nand，设置为 nor/nand 启动；
2. 上电与开发板的 usb device 口；安装驱动；
3. 下载内核：在 UBOOT 的串口菜单中输入 k；
4. 使用 dnw_100ask.exe 发送 uImage 文件；
5. uboot 即会自动接收、烧写 uImage 文件；
6. 下载文件系统：在 UBOOT 的串口菜单中输入 y；
7. 使用 dnw_100ask.exe 发送 fs_qtopia.yaffs2 文件；
8. uboot 即会自动接收、烧写根文件系统；
9. . 输入 q 退出 UBOOT 串口菜单，执行命令删除参数分区：nand erase params；
10. . 重启(对于 QT 文件系统，第一次重启时会要求你较准触摸屏)；

(如果触摸不准，可以等系统启动后在串口执行：rm /etc/pointercal 然后重启再次较准)；

第 007 课 裸机开发步骤和工具使用(免费))

001 节_裸机开发步骤简介

回忆大学在 VC6.0 上面学习 C 语言，操作大致流程如下：

x86 平台，Windows 应用程序：建立工程/项目，编辑源码，编译代码（编译器：VC6.0），运行。

在 Ubuntu 上学习 C 语言，操作大致流程如下：

x86 平台，Ubuntu 应用程序：编辑源码，编译代码（编译器：gcc），运行。

两者是几乎一致的。

名词解释：

- gcc：

gcc 是 linux 系统下主要的编译软件， gcc 的全称是 GNU 编译器套件（GNU Compiler Collection）， 除了可以编译 c 语言开发的程序外，还可以开发 C++、Java 等多个语言的程序。 GCC 的初衷是为 GNU 操作系统专门编写的一款编译器。

使用 gcc 的原因：功能强大、稳定、开源免费。

1）`gcc -v` 查看 gcc 的版本，从而验证了 gcc 编译器正常。

不论你使用哪一个版本的 gcc，只有后面能够正常编译就可以了。

2）`gcc -o 输出文件名 源文件` o:output

- 交叉编译模式：

通常编译嵌入式程序的平台成为宿主机（如：PC的ubuntu系统，CPU架构为X86架构）， 运行嵌入式程序的平台成为目标机（如：某款ARM开发板，CPU架构为ARM架构）。 在X86平台编辑和和编译器arm-linux-gcc编译ARM架构的程序， 两者属于不同的架构平台，从而属于交叉编译模式。 然后将程序烧写到ARM开发板中（下载方式有：JTAG、USB、SD卡、网络等多种方式）， 然后在开发板上运行该程序。

- 编辑器:

推荐使用windows平台的source insight和notepad。 source insight在查看代码、编辑代码等功能时非常好用。

- 编译器

推荐使用arm-linux-gcc arm-linux-gcc是基于linux平台的arm编译器。它是开源免费的编译器。 arm-linux-gcc功能强大、稳定、支持的arm芯片众多、更新速度快。

入门误区：使用ads、MDK
ads（停止更新）、MDK，是windows平台的编译器，功能较弱。
1）只适合个人或者小团队的开发，不适合于中型、大型团队的协作开发。
2）是收费软件。

002节_source insight代码查看工具的使用

编辑器
推荐使用windows平台的source insight和notepad。
source insight在查看代码、编辑代码等功能时非常好用。
source insight针对:c文件、h文件、ARM汇编文件
notepad针对：其他文件，如Makefile(简单理解为：指明了要编译哪些源文件、指定编译后的输出文件名)

source insight是收费软件，但是这个软件可以免费35天。 版本：3.5
安装包：自己百度一下"source insight3.5"

source insight常用设置：
1.增加文件类型

options-->documents options 中，将 c source file 的文件过滤设置为 *.c;*.h;*.S

2.新建项目(新建工程)

project-->new project，然后将相应的文件添加到该项目中。

3.字体的设置(解决中文乱码)

options-->documents options 中，将 screen fonts 设置为：宋体、常规、12、GB2312.

使用 alt+F12 来调整字体的等宽。

4.说明一下 source insight 的窗口

项目窗口：指明该项目中有哪些文件。
主窗口：显示当前打开文件的文件内容。
符号窗口：显示了当前打开文件的函数名、结构体名、宏定义等等。
上下文窗口：光标放在某个函数上(变量上、宏定义上)，在下面的上下文窗口就可以看到相应的定义。

5.显示行号

view-->line number

6.快速跳转到某一行

工具栏中有一个 go to line。

7.高亮关键词

光标放在关键词上，然后右键菜单--->highlight word

8.查看函数定义位置

光标放在函数上，然后右键菜单--->jump to defintion，或者 ctrl + 鼠标左键。
go back (alt + ,)回到上一次光标所在位置，或者使用工具栏中的向左箭头。

go forward(alt + .)回到下一次光标所在位置，或者使用工具栏中的向右箭头。

9. 查找

右键菜单-->lookup reference，或者 ctrl + / 。

003 节_notepad 编辑器和二进制文件查看工具

下面介绍两个免费的工具： 文本编辑器工具 Notepad++和二进制查看工具 Free Hex Editor Neo。

Notepad++使用技巧：

1. 按住 Ctrl+鼠标滚轮，可以缩放文本内容大小；
2. 查看 c 语言文件时，双击变量，即可在高亮整个文本中的该变量；
3. 查看 c 语言文件时，光标放在任一括号处，即可高亮整个括号范围；
4. 快捷键“Ctrl+F”，打开查找窗口，可搜索所需字符；

Free Hex Editor Neo 使用技巧：

1. 打开 hex 二进制文件，默认每行显示 16 个数，即 0x00-0x0f;
2. 在菜单栏的 View->Offset 可设置左边偏移的显示进制；
3. 在菜单栏的 View->Display As 可设置当前数据的显示进制；
4. 在菜单栏的 View->Columns As 可设置当前每行显示多少数据；
5. 在菜单栏的 View->Group By 可设置当前数据显示的数据类型；

第 008 课 第 1 个 ARM 裸板程序及引申(部分免费)

第 001 节_辅线 1_硬件知识_LED 原理图

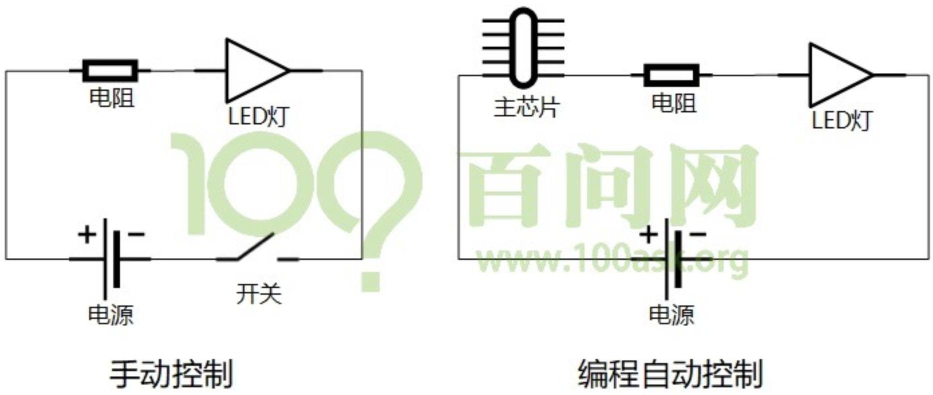
当我们学习 C 语言的时候，我们会写个 Hello 程序。那当我们写 ARM 程序，也该有一个简单的程序引领我们入门，这个程序就是点亮 LED。
我们怎样去点亮一个 LED 呢？ 分为三步：

1. 看原理图，确定控制 LED 的引脚;
2. 看主芯片的芯片手册，确定如何设置控制这个引脚;
3. 写程序;



它们长得完全不一样，因此我们在原理图中将它抽象出来。

点亮 LED 需要通电源，同时为了保护 LED，加个电阻减小电流。 控制 LED 灯的亮灭，可以手动开关 LED，但在电子系统中，不可能让人来控制开关，通过编程，利用芯片的引脚去控制开关。

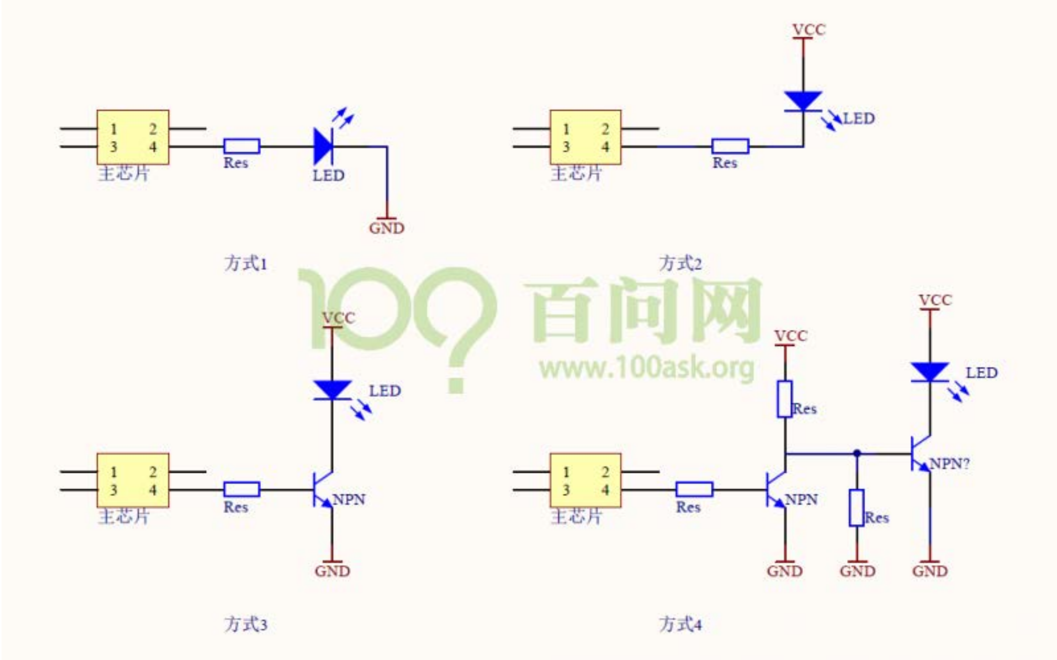


LED 的驱动方式，常见的有四种。

- 方式 1：使用引脚输出 3.3V 点亮 LED，输出 0V 熄灭 LED。
- 方式 2：使用引脚拉低到 0V 点亮 LED，输出 3.3V 熄灭 LED。

有的芯片为了省电等原因，其引脚驱动能力不足，这时可以使用三极管驱动。

- 方式 3：使用引脚输出 1.2V 点亮 LED，输出 0V 熄灭 LED。
- 方式 4：使用引脚输出 0V 点亮 LED，输出 1.2V 熄灭 LED。



由此，主芯片引脚输出高电平/低电平，即可改变 LED 状态，而无需关注 GPIO 引脚输出的是 3.3V 还是 1.2V。 所以简称输出 1 或 0：

- 逻辑 1-->高电平
- 逻辑 0-->低电平

第 002 节_辅线 1_硬件知识_S3C2440 启动流程与 GPIO 操作

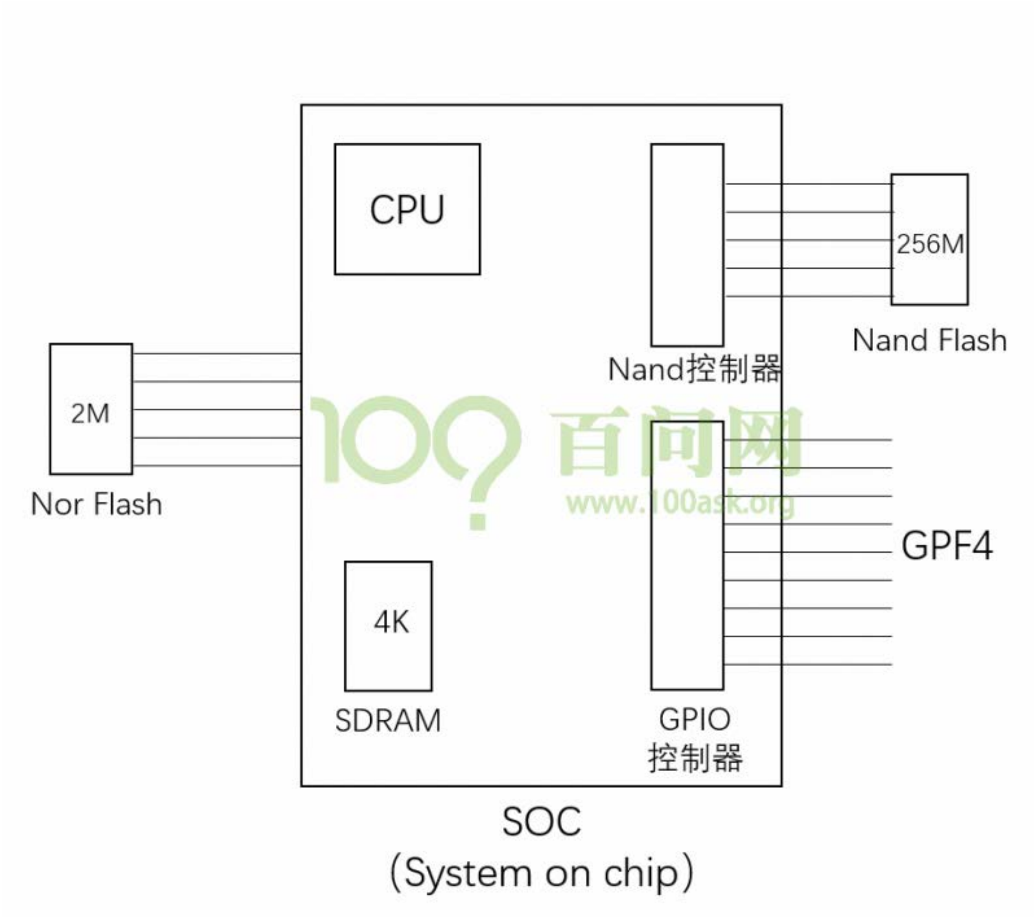
在原理图中，同名的 Net 表示是连在一起的。

怎么样 GPF4 怎么输出 1 或 0?

1. 配置为输出引脚；
2. 设置状态；

因此，设置 GPFCON[9:8]=0b01,即 GPF4 配置为输出；
设置 GPFDAT[4]=1 或者 0，即输出高电平或低电平；

S3C2440 框架：



S3C2440 启动流程：

- Nor 启动：

Nor Flash 的基地址为 0，片内 RAM 地址为 0x4000 0000;
CPU 读出 Nor 上第 1 个指令（前 4 字节），执行;
CPU 继续读出其它指令执行。

- Nand 启动：

片内 4k RAM 基地址为 0，Nor Flash 不可访问;
2440 硬件把 Nand 前 4K 内容复制到片内的 RAM，然后 CPU 从 0 地址取出第 1 条指令执行。

第 003 节_编写第 1 个程序点亮 LED

在开始写第 1 个程序前，先了解一些概念。
2440 是一个 SOC，它里面的 CPU 有 R1、R2、R3……等 寄存器;
它里面的 GPIO 控制器也有很多寄存器，如 GPFCON、GPFDAT。

这两个寄存器是有差异的，在写代码的时候，CPU 里面的寄存器可以直接访问，其它的寄存器要以地址进行访问。

把 GPF4 配置为输出，需要把 0x100 写入 GPFCON 这个寄存器，即写到 0x5600 0050 上；

把 GPF4 输出 1，需要把 0x10 写到地址 0x5600 0054 上；

把 GPF4 输出 0，需要把 0x00 写到地址 0x5600 0054 上；

这里的写法会破坏寄存器的其它位，其它位是控制其它引脚的，为了让第一个裸板程序尽可能的简单，才简单粗暴的这样处理。

写程序需要用到几条汇编代码：

①LDR (load):读寄存器

举例：`LDR R0,[R1]`

假设 R1 的值是 x，读取地址 x 上的数据（4 字节），保存到 R0 中；

②STR (store):写寄存器

举例：`STR R0,[R1]`

假设 R1 的值是 x，把 R0 的值写到地址 x（4 字节）；

③B 跳转

④MOV (move)移动，赋值 举例 1：`MOV R0,R1` 把 R1 的值赋值给 R0;

举例 2：`MOV R0,#0x100` 把 0x100 赋值给 R0,即 R0=0x100;

⑤LDR

举例：`LDR R0,=0x12345678` 这是一条伪指令，即实际中并不存在这个指令，他会被拆分成几个真正的 ARM 指令，实现一样的效果。 最后结果是 R0=0x12345678。

为什么会引入伪指令？

在 ARM 的 32 位指令中，有些字节表示指令，有些字节表示数据，因此表示数据的没有 32 位，不能表示一个 32 位的任意值，只能表示一个较小的简单值，这个简单值称为立即数。引入伪指令后，利用 LDR 可以为 R0 赋任意大小值，编译器会自动拆分成真正的的指令，实现目的。

有了前面 5 个汇编指令的基础，我们就可以写代码了。

第一个程序只能是汇编，以前你们可能写过单片机程序，一上来就写 main()函数，那是编译器帮你封装好了。

第一个 LED 程序代码如下：

```
/*
 * 点亮LED1: gpf4
 */

.text
.global _start

_start:

/* 配置GPF4为输出引脚

 * 把0x100写到地址0x56000050

 */
   ldr r1, =0x56000050
   ldr r0, =0x100      /* mov r0, #0x100 */
   str r0, [r1]


/* 设置GPF4输出高电平

 * 把0写到地址0x56000054

 */
   ldr r1, =0x56000054
   ldr r0, =0   /* mov r0, #0 */
   str r0, [r1]

   /* 死循环 */
halt:
   b halt
```

将代码上传到服务器，先编译：

arm-linux-gcc -c -o led_on.o led_on.s ;

再链接：

arm-linux-ld -Ttext 0 led_on.o -o led_on.elf ;

生成bin文件：

arm-linux-objcopy -O binary -S led_on.elf led_on.bin ;

以上的命令，要是我们每次都输入会容易输错，因此我们把他们写到一个文件里，这个文件就叫 Makefile. 关于 Makefile 以后会讲。本次所需的 Makefile 如下：

```
all:
	arm-linux-gcc -c -o led_on.o led_on.S
	arm-linux-ld -Ttext 0 led_on.o -o led_on.elf
	arm-linux-objcopy -O binary -S led_on.elf led_on.bin
clean:
	rm *.bin *.o *.elf
```

以后只需要 使用 make 命令进行编译， make clean 命令进行清理。
最后烧写到开发板上，即可看到只有一个 LED 亮，符合我们预期。

第 004 节_汇编与机器码

前面介绍过伪指令，伪指令是实际不存在的 ARM 命令，编译器在编译时转换成存在的 ARM 指令。我们代码中的 `ldr r1, =0x56000050` 这条伪指令的真实指令时什么呢?
我们可以通过反汇编来查看。

在前面的 Makefile 中加上：

```
arm-linux-objdump -D led_on.elf > led_on.dis
```

上传服务器，编译。
生成的 led_on.dis 就是反汇编文件。led_on.dis 如下：

```
led_on.elf:     file format elf32-littlearm


Disassembly of section .text:


00000000 <_start>:

   0:	e59f1014 	ldr	r1, [pc, #20]	; 1c <.text+0x1c>

   4:	e3a00c01 	mov	r0, #256	; 0x100

   8:	e5810000 	str	r0, [r1]
```

```
   c:      e59f100c        ldr     r1, [pc, #12] ; 20 <.text+0x20>
  10:      e3a00000        mov     r0, #0 ; 0x0
  14:      e5810000        str     r0, [r1]

00000018 <halt>:
  18:      eafffffe        b       18 <halt>
  1c:      56000050        undefined
  20:      56000054        undefined
```

第一列是地址，第二列是机器码，第三列是汇编；

在反汇编文件里可以看到，`ldr r1, =0x56000050` 被转换成 `ldr r1, [pc, #20]`，`pc+20` 地址的值为 0x56000050，通过这种方式为 r1 赋值。 对于立即数 0x100 而言，`ldr r0,=0x100` 即是转换成了 `mov r0,#256`；

在 2440 这个 SOC 里面，R0-R15 都在 CPU 里面，其中：

R13　别名：sp　（Stack Pointer）栈指针
R14　别名：lr　（Link Register）返回地址
R15　别名：pc　（program Counter）程序计数器=当前指令+8

为什么　PC=当前指令+8？

ARM 指令采用流水线机制，当前执行地址 A 的指令，已经在对地址 A+4 的指令进行译码，已经在读取地址 A+8 的指令，其中 A+8 就是 PC 的值。

C/汇编（给人类方便使用的语言）————编译器————>bin，含有机器码（给 CPU 使用）

第 005 节_编程知识_进制

17 个苹果，有 4 种表示方式，它们表示同一个数



- 计算验证：
  十进制：17=1x10^1 + 7x10^0;
  二进制：17=1x2^4 + 0x2^3 + 0x2^2 + 0x2^1 + 1x2^0;
  八进制：17=2x8^1 + 1x8^0;
  十六进制：17=1x16^1 + 1x16^0;
- 为何引入二进制？

在硬件角度看，晶体管只有两个状态：on 是 1,off 是 0； 数据使用多个晶体管进行表示，用二进制描述，吻合硬件状态。

- 为何引入八进制？

将二进制的三位作为一组，把这一组作为一位进行表示，就是八进制。

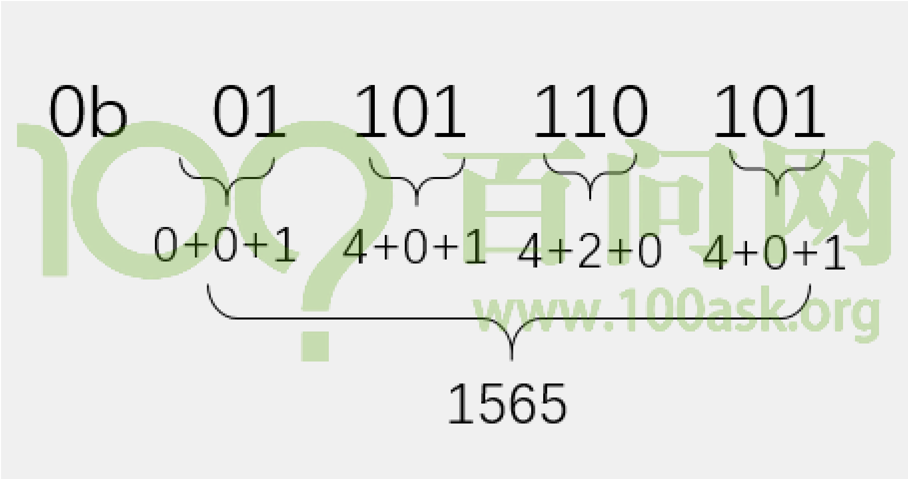
- 为何引入十六进制？

将二进制的四位作为一组，把这一组作为一位进行表示，就是十六进制。八进制和十六进制方便我们描述，简化了长度。

如何快速的转换 2/8/16 进制： 首先记住 **8 4 2 1 ——>二进制权重**

- 举例 1：

将二进制 0b01101110101 转换成八进制： 将二进制从右到左，每三个分成

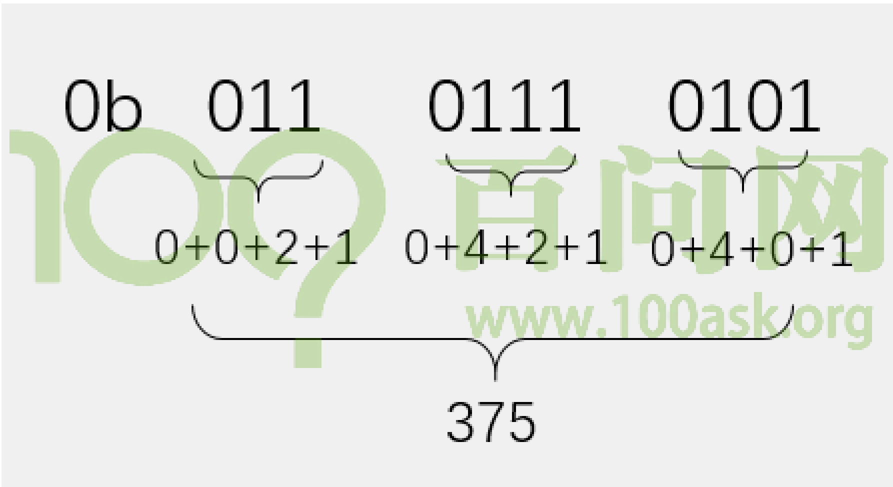


一组：

结果就是 1565；

- 举例 2：

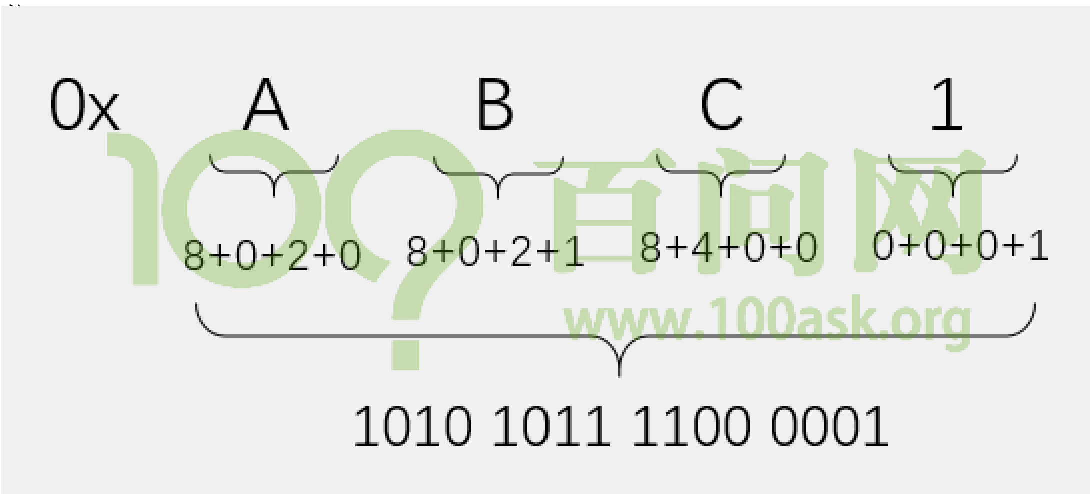
将二进制 0b01101110101 转换成十六进制： 将二进制从右到左，每四个分成一组：



结果就是 375；

- 举例 3：

将十六进制 0xABC1 转换成二进制： 将十六进制从右到左，每个分成四



结果就是 1010 1011 1100 0001；

在 C 语言中怎么表示这些进制呢？

十进制： int a = 96;
八进制： int a = 0140;//0 开头
十六进制： int a = 0x60;//0x 开头

用 0b 开头表示二进制，约定俗成的规定。

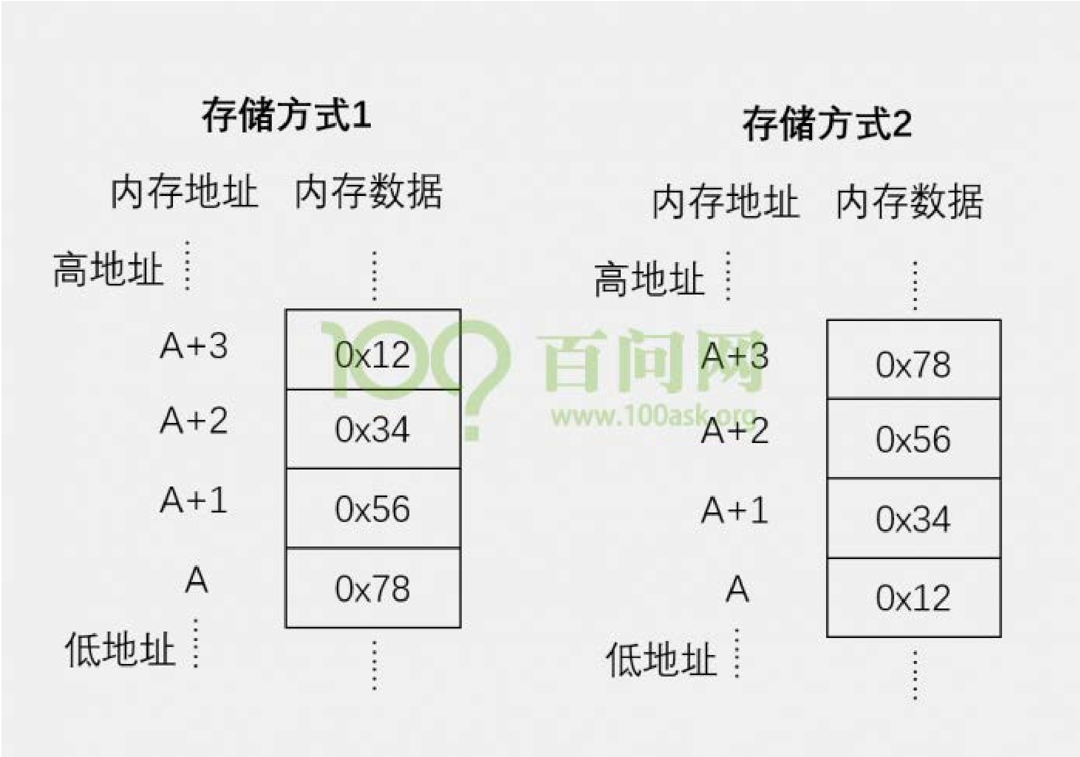
第 006 节_编程知识_字节序_位操作

- 字节序：

假设 `int a = 0x12345678;`

前面说了 16 进制每位是 4 个字节，在内存中，是以 8 个字节作为 1byte 进行存储的，因此 0x12345678 中每两位作为 1byte,其中 0x78 是低位，0x12 是高位。

在内存中的存储方式有两种：



0x12345678 的低位（0x78）存在低地址，即方式 1，叫做小字节序(Little endian)；

0x12345678 的高位（0x12）存在低地址，即方式 2，叫做大字节序(Big endian)；

一般的 arm 芯片都是小字节序，对于 2440 可以设置某个寄存器，让整个系统使用大字节序或小字节序，它默认使用小字节序。

- 位操作：

1. 移位

左移：

int a = 0x123; int b = a<<2;--> b=0x48C

右移：

int a = 0x123; int b = a>>2;--> b=0x48

左移是乘 4，右移是除 4；

2. 取反 原来问 0 的位变 1，原来为 1 的位变 0；

int a = 0x123; int b = ~a;a=2

3. 位与

```
1 & 1 = 1
1 & 0 = 0
0 & 1 = 0
0 & 0 = 0
int a = 0x123; int b = 0x456; int c = a&b;--> c=0x2
```

4. 位或

```
1 | 1 = 1
1 | 0 = 1
0 | 1 = 1
0 | 0 = 0
int a = 0x123; int b = 0x456; int c = a|b;--> c=0x577
```

5. 置位 把 a 的 bit7、8 置位（变为 1）

```
int a = 0x123; int b = a|(1<<7)|(1<<8);--> c=0x1a3
```

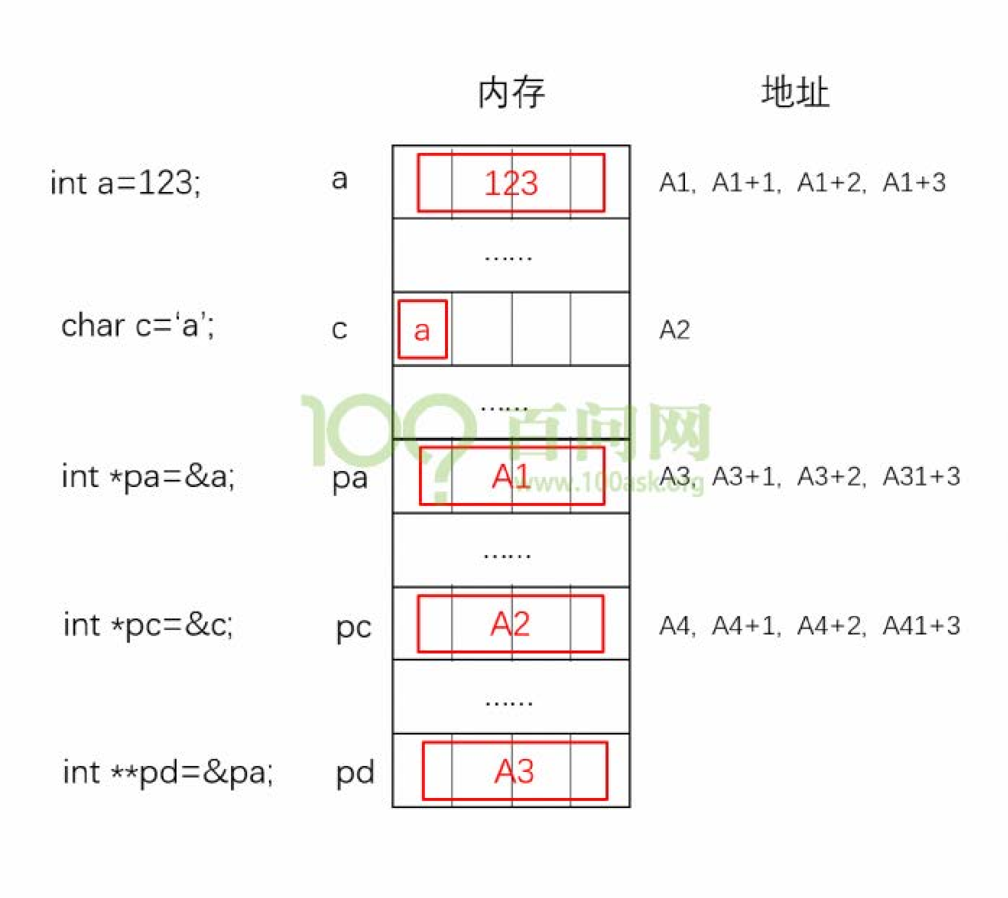
6. 清位 把 a 的 bit7、8 清位（变为 0）

```
int a = 0x123; int b = (a& ~(1<<7))&(~(1<<8));--> c=0x23
```

置位和清位在后面寄存器的操作中，会经常使用。

第 007 节_编写 C 程序控制 LED

C 语言的指针操作：
①所有的变量在内存中都有一块区域；
②可以通过变量/指针操作内存；




```
TYPE *p = val1;
*p = val2;
```

把 val2 写入地址 val1 的内存中，写入 `sizeof(TYPE)` 字节;

```
TYPE *p = addr;
*p = val;
```

把 val 写入地址 addrd 的内存，，写入 `sizeof(TYPE)` 字节;

a. 我们写出了 main 函数，谁来调用它? b. main 函数中变量保存在内存中，这个内存地址是多少? 答：我们还需要写一个汇编代码，给 main 函数设置内存，调用 main 函数

led.c 源码:

```
int main()
{
    unsigned int *pGPFCON = (unsigned int *)0x56000050;
    unsigned int *pGPFDAT = (unsigned int *)0x56000054;
```

```
    /*配置GPF4为输出引脚*/
    *pGPFCON = 0x100;

    /*配置GPF4输出0*/
    *pGPFDAT = 0;

    return 0;
}
```

start.S源码：

```
.text
.global _start
_start:
    /*设置内存：sp栈*/

    ldr sp,=4096 /*nand启动*/

//  ldr sp, =0x40000000 /*nor启动*/


    /*调用main*/

    bl main
halt:
    b halt
```

Makefile源码：

```
all:
    arm-linux-gcc -c -o led.o led.c
    arm-linux-gcc -c -o start.o start.S
    arm-linux-ld -Ttext 0 start.o led.o -o led.elf
    arm-linux-objcopy -O binary -S led.elf led.bin
    arm-linux-objdump -D led.elf > led.dis
clean:
    rm *.bin *.o *.elf *.dis
```

最后将上面三个文件放入Ubuntu主机编译，然后烧写到开发板即可。

第008节_几条汇编指令

_bl_add_sub_ldm_stm

⑥ADD/SUB 加法/减法

举例 1：

add r0,r1,#4

效果为

r0=r1+4;

举例 2：

sub r0,r1,#4

效果为

r0=r1-4;

举例 3：

sub r0,r1,r2

效果为

r0=r1-r2;

⑦BL (Brarch and Link)带返回值的跳转 跳转到指定指令，并将返回地址（下一条指令）保存在 lr 寄存器；

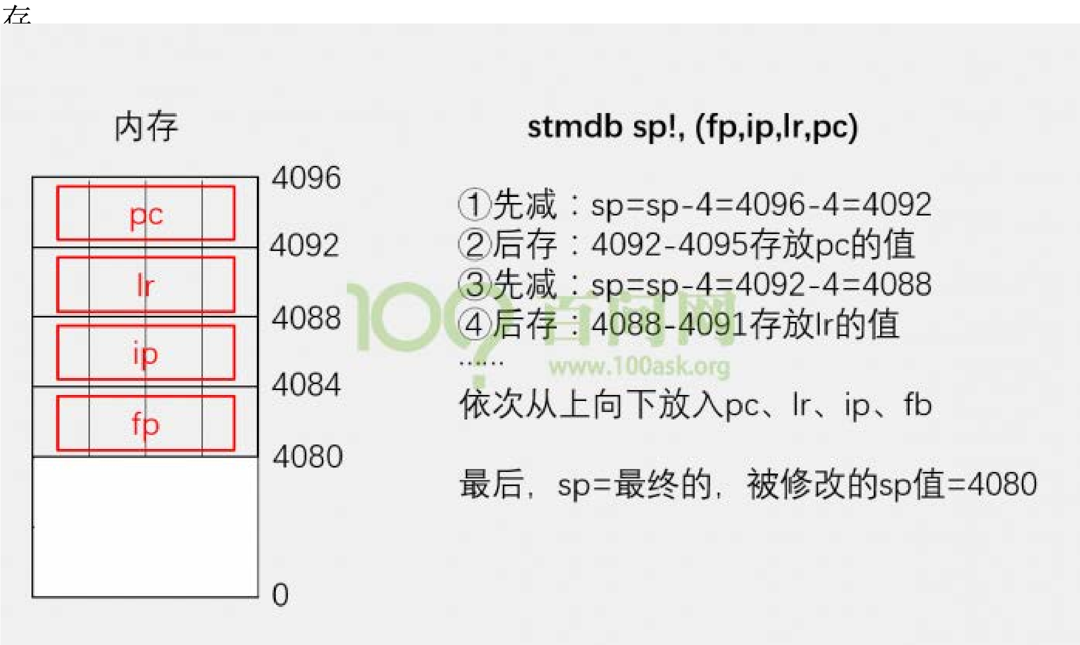
⑧LDM/STM 读内存，写入多个寄存器/把多个寄存器的值写入内存

可搭配的后缀有 过后增加(Increment After)、预先增加(Increment Before)、过后减少(Decrement After)、预先减少(Decrement Before);

举例 1：

stmdb sp!, (fp,ip,lr,pc)

假设 Sp=4096。 db 意思是先减后存，按 高编号寄存器存在高地址
存



举例 2：

ldmia sp, (fp,ip,pc)



009 节_解析 C 程序的内部机制

003_led.c 内部机制分析：
start.S:
①设置栈；
②调用 main，并把返回值地址保存到 lr 中；

led.c 的 main()内容：
①定义 2 个局部变量；
②设置变量；
③return 0；

问题：
①为什么要设置栈？
因为 c 函数要用。

②怎么使用栈？
a.保存局部变量；
b.保存 lr 等寄存器；

③调用者如何传参数给被调用者？
④被调用者如何传返回值给调用者？
⑤怎么从栈中恢复那些寄存器？

在 arm 中有个 ATPCS 规则，约定 r0-r15 寄存器的用途。
r0-r3:调用者和被调用者之间传参数；
r4-r11:函数可能被使用，所以在函数的入口保存它们，在函数的出口恢复它们；

下面分析个实例 start.S:

```
.text
.global _start

_start:

    /* 设置内存: sp 栈 */

    ldr sp, =4096  /* nand 启动 */

// ldr sp, =0x40000000+4096  /* nor 启动 */


    /* 调用 main */

    bl main

halt:
    b halt
```

led.c:

```c
int main()
{
    unsigned int *pGPFCON = (unsigned int *)0x56000050;
    unsigned int *pGPFDAT = (unsigned int *)0x56000054;

    /* 配置 GPF4 为输出引脚 */
    *pGPFCON = 0x100;

    /* 设置 GPF4 输出 0 */
    *pGPFDAT = 0;

    return 0;
}
```

将前面的程序反汇编得到 led.dis 如下：

```
led.elf:     file format elf32-littlearm

Disassembly of section .text:

00000000 <_start>:
   0:	e3a0da01 	mov	sp, #4096	; 0x1000
   4:	eb000000 	bl	c <main>

00000008 <halt>:
   8:	eafffffe 	b	8 <halt>

0000000c <main>:
   c:	e1a0c00d 	mov	ip, sp
  10:	e92dd800 	stmdb	sp!, {fp, ip, lr, pc}
  14:	e24cb004 	sub	fp, ip, #4	; 0x4
  18:	e24dd008 	sub	sp, sp, #8	; 0x8
  1c:	e3a03456 	mov	r3, #1442840576	;
0x56000000
  20:	e2833050 	add	r3, r3, #80	; 0x50
  24:	e50b3010 	str	r3, [fp, #-16]
  28:	e3a03456 	mov	r3, #1442840576	;
0x56000000
  2c:	e2833054 	add	r3, r3, #84	; 0x54
  30:	e50b3014 	str	r3, [fp, #-20]
  34:	e51b2010 	ldr	r2, [fp, #-16]
  38:	e3a03c01 	mov	r3, #256	; 0x100
  3c:	e5823000 	str	r3, [r2]
```

```
 40:     e51b2014     ldr    r2, [fp, #-20]
 44:     e3a03000     mov    r3, #0 ; 0x0
 48:     e5823000     str    r3, [r2]
 4c:     e3a03000     mov    r3, #0 ; 0x0
 50:     e1a00003     mov    r0, r3
 54:     e24bd00c     sub    sp, fp, #12   ; 0xc
 58:     e89da800     ldmia  sp, {fp, sp, pc}
Disassembly of section .comment:

00000000 <.comment>:
  0:     43434700     cmpmi  r3, #0 ; 0x0
  4:     4728203a     undefined
  8:     2029554e     eorcs  r5, r9, lr, asr #10
  c:     2e342e33     mrccs  14, 1, r2, cr4, cr3, {1}
 10:     Address 0x10 is out of bounds.
```

分析上面的汇编代码：
开发板上电后，将从 0 地址开始执行，即开始执行

mov sp, #4096:设置栈地址在 4k RAM 的最高处，sp=4096;
bl c <main>:调到 c 地址处的 main 函数，并保存下一行代码地址到 lr，即 lr=8;
mov ip, sp:给 ip 赋值 sp 的值，ip=sp=4096
stmdb sp!, {fp, ip, lr, pc}:按高编号寄存器存在高地址，依次将 pc、lr、ip、fp 存入 sp-4 中;
sub fp, ip, #4: fp 的值为 ip-4=4096-4=4092;
sub sp, sp, #8:sp 的值为 sp-8=(4096-4x4)-8=4072;
mov r3, #1442840576:r3 赋值 0x5600 0000;
add r3, r3, #80:r3 的值加 0x50,即 r3=0x5600 0050;
str r3, [fp, #-16]:r3 存入[fp-16]所在的地址，即地址 4076 处存放 0x5600 0050;
mov r3, #1442840576: r3 赋值 0x5600 0000;
add r3, r3, #84:r3 的值加 0x54,即 r3=0x5600 0054;
str r3, [fp, #-20]:r3 存入[fp-20]所在的地址，即地址 4072 处存放 0x5600 0054;
ldr r2, [fp, #-16]: r2 取[fp-16]地址处的值，即[4076]地址的值，r2=0x5600 0050;
mov r3, #256:r3 赋值为 0x100;
str r3, [r2]: 将 r3 写到 r2 内容所对应的地址，即 0x5600 0050 地址处的值为 0x100;;对应 c 语言*pGPFCON = 0x100;;
ldr r2, [fp, #-20]:r2 取[fp-20]地址处的值，即[4072]地址的值，r2=0x5600 0054;
mov r3, #0:r3 赋值为 0x00;

str r3, [r2]:将 r3 写到 r2 内容所对应的地址，即 0x5600 0054 地址处的值为 0x00;对应 c 语言*pGPFDAT = 0;

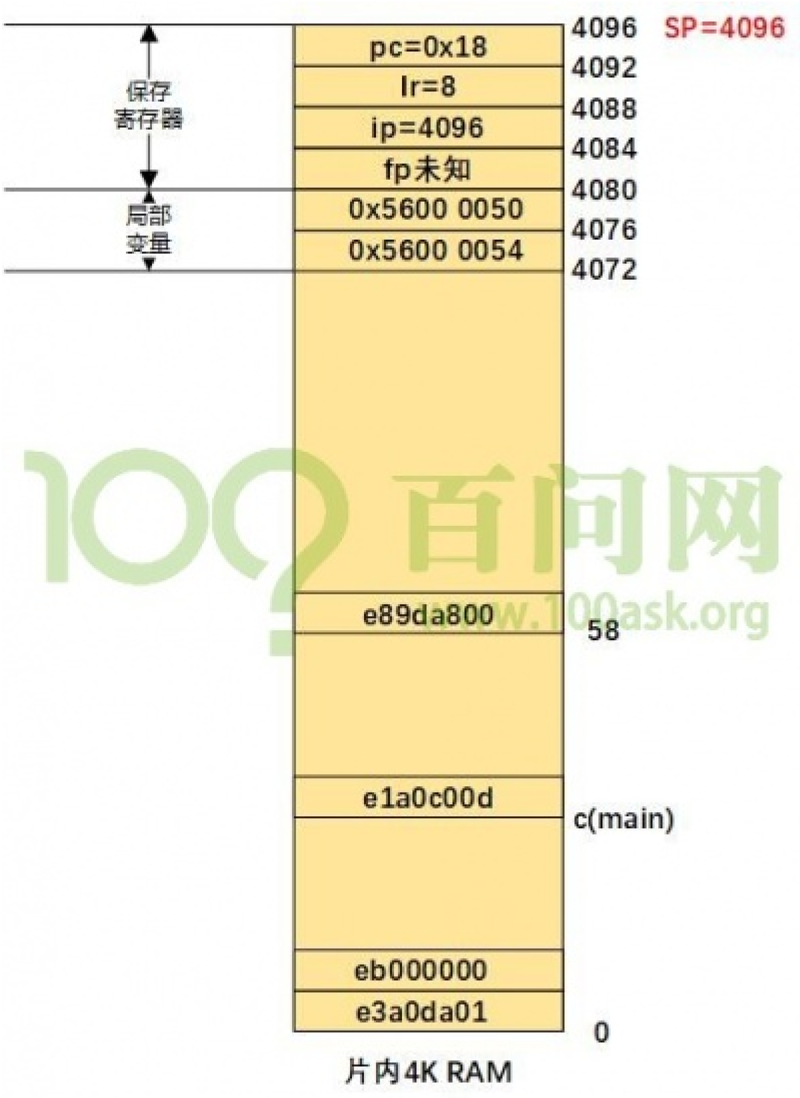
mov r3, #0:r3 赋值为 0x00;

mov r0, r3: r0=r3=0x00;

sub sp, fp, #12:sp=fp-12=4092-12=4080;

ldmia sp, {fp, sp, pc}:从栈中恢复寄存器，fp=4080 地址处的值=原来的 fp,sp=4084 地址处的值=4096，pc=4088 地址处的值=8，随后调到 0x08 地址处继续执行。

过程中的内存数据情况:



前面那个例子，汇编调用 main.c 并没有传递参数，这里修改下 c 程序，让其传递参数。

start.S:

```
.text
.global _start
```

```
_start:

    /* 设置内存: sp 栈 */

    ldr sp, =4096  /* nand 启动 */

//  ldr sp, =0x40000000+4096  /* nor 启动 */

    mov r0, #4
    bl led_on

    ldr r0, =100000
    bl delay

    mov r0, #5
    bl led_on

halt:
    b halt
```

led.c:

```c
void delay(volatile int d)
{
    while (d--);
}

int led_on(int which)
{
    unsigned int *pGPFCON = (unsigned int *)0x56000050;
    unsigned int *pGPFDAT = (unsigned int *)0x56000054;

    if (which == 4)
    {
        /* 配置 GPF4 为输出引脚 */

        *pGPFCON = 0x100;
    }
    else if (which == 5)
    {
        /* 配置 GPF5 为输出引脚 */

        *pGPFCON = 0x400;
```

```
    }

    /* 设置GPF4/5输出0 */
    *pGPFDAT = 0;

    return 0;
}
```

led.elf:

```
led.elf:     file format elf32-littlearm

Disassembly of section .text:

00000000 <_start>:
   0:      e3a0da01     mov     sp, #4096     ; 0x1000
   4:      e3a00004     mov     r0, #4 ; 0x4
   8:      eb000012     bl      58 <led_on>
   c:      e59f000c     ldr     r0, [pc, #12]; 20
<.text+0x20>
  10:      eb000003     bl      24 <delay>
  14:      e3a00005     mov     r0, #5 ; 0x5
  18:      eb00000e     bl      58 <led_on>

0000001c <halt>:
  1c:      eafffffe     b       1c <halt>
  20:      000186a0     andeq  r8, r1, r0, lsr #13

00000024 <delay>:
  24:      e1a0c00d     mov    ip, sp
  28:      e92dd800     stmdb  sp!, {fp, ip, lr, pc}
  2c:      e24cb004     sub    fp, ip, #4    ; 0x4
  30:      e24dd004     sub    sp, sp, #4    ; 0x4
  34:      e50b0010     str    r0, [fp, #-16]
  38:      e51b3010     ldr    r3, [fp, #-16]
  3c:      e2433001     sub    r3, r3, #1    ; 0x1
  40:      e50b3010     str    r3, [fp, #-16]
  44:      e51b3010     ldr    r3, [fp, #-16]
  48:      e3730001     cmn    r3, #1 ; 0x1
  4c:      0a000000     beq    54 <delay+0x30>
  50:      eafffff8     b      38 <delay+0x14>
  54:      e89da808     ldmia  sp, {r3, fp, sp, pc}

00000058 <led_on>:
```

```
    58:     e1a0c00d     mov    ip, sp
    5c:     e92dd800     stmdb  sp!, {fp, ip, lr, pc}
    60:     e24cb004     sub    fp, ip, #4    ; 0x4
    64:     e24dd00c     sub    sp, sp, #12   ; 0xc
    68:     e50b0010     str    r0, [fp, #-16]
    6c:     e3a03456     mov    r3, #1442840576     ;
0x56000000
    70:     e2833050     add    r3, r3, #80   ; 0x50
    74:     e50b3014     str    r3, [fp, #-20]
    78:     e3a03456     mov    r3, #1442840576     ;
0x56000000
    7c:     e2833054     add    r3, r3, #84   ; 0x54
    80:     e50b3018     str    r3, [fp, #-24]
    84:     e51b3010     ldr    r3, [fp, #-16]
    88:     e3530004     cmp    r3, #4 ; 0x4
    8c:     1a000003     bne    a0 <led_on+0x48>
    90:     e51b2014     ldr    r2, [fp, #-20]
    94:     e3a03c01     mov    r3, #256      ; 0x100
    98:     e5823000     str    r3, [r2]
    9c:     ea000005     b      b8 <led_on+0x60>
    a0:     e51b3010     ldr    r3, [fp, #-16]
    a4:     e3530005     cmp    r3, #5 ; 0x5
    a8:     1a000002     bne    b8 <led_on+0x60>
    ac:     e51b2014     ldr    r2, [fp, #-20]
    b0:     e3a03b01     mov    r3, #1024     ; 0x400
    b4:     e5823000     str    r3, [r2]
    b8:     e51b3018     ldr    r3, [fp, #-24]
    bc:     e3a02000     mov    r2, #0 ; 0x0
    c0:     e5832000     str    r2, [r3]
    c4:     e3a03000     mov    r3, #0 ; 0x0
    c8:     e1a00003     mov    r0, r3
    cc:     e24bd00c     sub    sp, fp, #12   ; 0xc
    d0:     e89da800     ldmia  sp, {fp, sp, pc}
  Disassembly of section .comment:

  00000000 <.comment>:
     0:     43434700     cmpmi  r3, #0 ; 0x0
     4:     4728203a     undefined
     8:     2029554e     eorcs  r5, r9, lr, asr #10
     c:     2e342e33     mrccs  14, 1, r2, cr4, cr3, {1}
    10:     Address 0x10 is out of bounds.
```

简单分析下反汇编：

mov sp, #4096:设置栈地址在 4k RAM 的最高处，sp=4096;

mov r0, #4:r0=4，作为参数;

bl 58 <led_on>:调到 58 地址处的 led_on 函数，并保存下一行代码地址到 lr，即 lr=8;在 led_on 中会使用到 r0;

ldr r0, [pc, #12]:r0=[pc+12]处的值=[c+12=20]的值=0x186a0=1000000，作为参数;

bl 24 <delay>:调用 24 地址处的 delay 函数，并保存下一行代码地址到 lr，即 lr=24;在 delay 中会使用到 r0;

mov r0, #5：r0=5，作为参数;

bl 58 <led_on>：调到 58 地址处的 led_on 函数，并保存下一行代码地址到 lr，即 lr=58;在 led_on 中会使用到 r0;

010 节_完善 LED 程序_编写按键程序

在上一节视频里，我们编写的程序代码是先点亮 led1，然后延时一会，再点亮 led2，进入死循环。

但在开发板上的实际效果是 led1 先亮，延时一会，led2 再亮，然后一会之后，led1 再次亮了。

这和我们的设计的代码流程不吻合，这是因为 2440 里面有个看门狗定时器，开发板上电后，需要在一定时间内“喂狗”（设置相应的寄存器），否则就会重启开发板。

之所以这样设计，是为了让芯片出现死机时，能够自己复位，重新运行。

这里我们写个 led 灯循环的程序，步骤如下：

1. 这里暂时用不到看门狗，先关闭看门狗，从参考手册可知，向 0x53000000 寄存器写 0 即可关闭看门狗;
2. 设置内存的栈，通过写读操作来判断是 Nand Flash 还是 Nor Flash;
3. 设置 GPFCON 让 GPF4/5/6 配置为输出引脚;
4. 循环点灯，依次设置 GPFDAT 寄存器;

完整代码如下：

```
.text
.global _start

_start:

    /* 关闭看门狗 */
    ldr r0, =0x53000000
    ldr r1, =0
    str r1, [r0]
```

```
    /* 设置内存: sp 栈 */
    /* 分辨是nor/nand启动
     * 写0到0地址, 再读出来
     * 如果得到0, 表示0地址上的内容被修改了, 它对应ram, 这就
是nand启动
     * 否则就是nor启动
     */
    mov r1, #0
    ldr r0, [r1] /* 读出原来的值备份 */
    str r1, [r1] /* 0->[0] */
    ldr r2, [r1] /* r2=[0] */
    cmp r1, r2   /* r1==r2? 如果相等表示是NAND启动 */
    ldr sp, =0x40000000+4096 /* 先假设是nor启动 */
    moveq sp, #4096  /* nand启动 */
    streq r0, [r1]   /* 恢复原来的值 */

    bl main

halt:
    b halt
```

led.c

```
void delay(volatile int d)
{
    while (d--);
}

int main(void)
{
    volatile unsigned int *pGPFCON = (volatile unsigned
int *)0x56000050;
```

```
	volatile unsigned int *pGPFDAT = (volatile unsigned
int *)0x56000054;
	int val = 0;  /* val: 0b000, 0b111 */
	int tmp;

	/* 设置GPFCON 让GPF4/5/6 配置为输出引脚 */
	*pGPFCON &= ~((3<<8) | (3<<10) | (3<<12));
	*pGPFCON |=  ((1<<8) | (1<<10) | (1<<12));

	/* 循环点亮 */
	while (1)
	{
		tmp = ~val;
		tmp &= 7;
		*pGPFDAT &= ~(7<<4);
		*pGPFDAT |= (tmp<<4);
		delay(100000);
		val++;
		if (val == 8)
			val =0;

	}

	return 0;
}
```

2440 里面有很多寄存器，如果每次对不同的寄存器进行查询和操作会很麻烦，因此可以先提前定义成宏，做成一个头文件，每次调用就行。

再举一个按键控制 LED 的程序，，步骤如下：

1. 这里暂时用不到看门狗，先关闭看门狗，从参考手册可知，向 0x53000000 寄存器写 0 即可关闭看门狗；
2. 设置内存的栈，通过写读操作来判断是 Nand Flash 还是 Nor Flash;
3. 设置 GPFCON 让 GPF4/5/6 配置为输出引脚;
4. 设置 3 个按键引脚为输入引脚;
5. 循环执行，读取按键引脚值，点亮对应的 led 灯；

完整代码如下：

```
#include "s3c2440_soc.h"

void delay(volatile int d)
```

```c
{
	while (d--);
}

int main(void)
{
	int val1, val2;

	/* 设置GPFCON让GPF4/5/6配置为输出引脚 */
	GPFCON &= ~((3<<8) | (3<<10) | (3<<12));
	GPFCON |=  ((1<<8) | (1<<10) | (1<<12));

	/* 配置3个按键引脚为输入引脚:
	 * GPF0(S2),GPF2(S3),GPG3(S4)
	 */
	GPFCON &= ~((3<<0) | (3<<4));  /* gpf0,2 */
	GPGCON &= ~((3<<6));  /* gpg3 */

	/* 循环点亮 */
	while (1)
	{
		val1 = GPFDAT;
		val2 = GPGDAT;

		if (val1 & (1<<0)) /* s2 --> gpf6 */
		{
			/* 松开 */
			GPFDAT |= (1<<6);
		}
		else
		{
			/* 按下 */
			GPFDAT &= ~(1<<6);
		}

		if (val1 & (1<<2)) /* s3 --> gpf5 */
		{
			/* 松开 */
```

```
                GPFDAT |= (1<<5);
            }
            else
            {
                /* 按下 */
                GPFDAT &= ~(1<<5);
            }

            if (val2 & (1<<3)) /* s4 --> gpf4 */
            {
                /* 松开 */
                GPFDAT |= (1<<4);
            }
            else
            {
                /* 按下 */
                GPFDAT &= ~(1<<4);
            }


        }

        return 0;
    }
```

第 009 课 gcc 和 arm-linux-gcc 和 Makefile

第 001 节_gcc 编译器 1_gcc 常用选项__gcc 编译过程详解

gcc 的使用方法

gcc [选项] 文件名

gcc 常用选项

选项	功能
-v	查看 gcc 编译器的版本，显示 gcc 执行时的详细过程
-o <file>	指定输出文件名为 file，这个名称不能跟源文件名同名
-E	只预处理，不会编译、汇编、链接 t
-S	只编译，不会汇编、链接

-c	编译和汇编，不会链接

一个 c/c++文件要经过预处理、编译、汇编和链接才能变成可执行文件。

（1）预处理

C/C++源文件中，以#开头的命令被称为预处理命令，如包含命令 **#include**、宏定义命令**#define**、条件编译命令**#if**、**#ifdef** 等。预处理就是将要包含(include)的文件插入原文件中、将宏定义展开、根据条件编译命令选择要使用的代码，最后将这些东西输出到一个**.i** 文件中等待进一步处理。

（2）编译

编译就是把 C/C++代码(比如上述的**.i** 文件)翻译成汇编代码。

（3）汇编

汇编就是将第二步输出的汇编代码翻译成符合一定格式的机器代码，在 Linux 系统上一般表现为 ELF 目标文件(OBJ 文件)。反汇编是指将机器代码转换为汇编代码，这在调试程序时常常用到。

（4）链接

链接就是将上步生成的 OBJ 文件和系统库的 OBJ 文件、库文件链接起来，最终生成了可以在特定平台运行的可执行文件。

hello.c(预处理)->hello.i(编译)->hello.s(汇编)->hello.o(链接)->hello

详细的每一步命令如下：

gcc -E -o hello.i hello.c

gcc -S -o hello.s hello.i

gcc -c -o hello.o hello.s

gcc -o hello hello.o

上面一连串命令比较麻烦，gcc 会对.c 文件默认进行预处理操作，使用-c 再来指明了编译、汇编，从而得到.o 文件，再将.o 文件进行链接，得到可执行应用程序。简化如下：

gcc -c -o hello.o hello.c

gcc -o hello hello.o

第 002 节_gcc 编译器 2_深入讲解链接过程

前面编译出来的可执行文件比源代码大了很多，这是什么原因呢？

我们从链接过程来分析，链接将汇编生成的 OBJ 文件、系统库的 OBJ 文件、库文件链接起来，crt1.o、crti.o、crtbegin.o、crtend.o、crtn.o 这些都是 gcc 加入的系统标准启动文件，它们的加入使最后出来的可执行文件相原来大了很多。

-lc：链接 libc 库文件，其中 libc 库文件中就实现了 printf 等函数。

gcc -v -nostdlib -o hello hello.o:

会提示因为没有链接系统标准启动文件和标准库文件，而链接失败。

这个-nostdlib 选项常用于裸机 bootloader、linux 内核等程序，因为它们不需要启动文件、标准库文件。

一般应用程序才需要系统标准启动文件和标准库文件。 裸机/bootloader、linux 内核等程序不需要启动文件、标准库文件。

动态链接使用动态链接库进行链接，生成的程序在执行的时候需要加载所需的动态库才能运行。

动态链接生成的程序体积较小，但是必须依赖所需的动态库，否则无法执行。

gcc -c -o hello.o hello.c
gcc -o hello_shared   hello.o

静态链接使用静态库进行链接，生成的程序包含程序运行所需要的全部库，可以直接运行，
不过静态链接生成的程序体积较大。

gcc -c -o hello.o hello.c
gcc -static -o hello_static hello.o

第 003 节_c 语言指针复习 1__指向 char 和 int 的指针

日常中，我们把笔记写到记事本中，记事本就相当于一个载体（存储笔记的内容）。
C 语言中有些变量，例如，char、int 类型的变量，它们也需要一个载体，来存储这些变量的值，这个载体就是内存。
比如我们的电脑内存有 4GB 内存，也就是 **4*1024*1024*1024=4294967296** 字节。

我们可以把整个内存想象成一串连续格子，每个格子(字节)都可以放入一个数据，如下图所示。



每一个小格子都有一个编号，小格子的编号从 0 开始，我们可以通过读取格子的编号，得到格子里面的内容。同理，我们根据内存的变量的地址，来获得其中的数据。
下面写个小程序进行测试，实例：

point_test.c

```
#include <stdio.h>

int main(int argc, char *argv[])
{
   printf("sizeof(char   ) = %d\n",sizeof(char   ));
   printf("sizeof(int    ) = %d\n",sizeof(int    ));
```

```c
    printf("sizeof(char  *) = %d\n",sizeof(char  *));
    printf("sizeof(char  **) = %d\n",sizeof(char **));

    return 0;
}
```

根据程序可以看出来，函数的功能是输出,char,int,char **类型所占据的字节数；

编译

gcc -o pointer_test pointer_test.c

运行应用程序：

./pointer_test

结果：（我用的是 64 位的编译器）

sizeof(char   ) = 1
sizeof(int   ) = 4
sizeof(char   *) = 8
sizeof(char **) = 8

可以看出在 64 位的机器中，用 8 个字节表示指针，我们可以测试一下用 32 位的机器编译

编译：

gcc -m32 -o pointer_test pointer_test.c   //加上**-m32**:编译成 32 位的机器码

编译可能会出现下面提示错误：

/usr/include/features.h:374:25: fatal error: sys/cdefs.h: No such file or directory

解决错误，安装 `lib32readline-gplv2-dev`，执行：

sudo apt-get install lib32readline-gplv2-dev

重新编译

gcc -m32 -o pointer_test pointer_test.c   //没有错误

运行生成的应用程序

./pointer_test

结果：

sizeof(char   ) = 1
sizeof(int   ) = 4
sizeof(char   *) = 4
sizeof(char **) = 4

可以看出编译成 32 位的机器码，指针就是用 4 个字节来存储的，

总结：

1. 所用变量不论是普通变量(char,int)还是指针变量，都存在内存中。
2. 所用变量都可以保存某些值。
3. 怎么使用指针?

取值

移动指针

实例 0

步骤一

```c
#include <stdio.h>

void test0()
{
    char c;
    char *pc;

    /*第一步 ： 所有变量都保存在内存中，我们打印一下变量的存储地址
*/
    printf("&c  =%p\n",&c);
    printf("&pc =%p\n",&pc);

}

int main(int argc, char *argv[])
{
    printf("sizeof(char   ) = %d\n",sizeof(char   ));
    printf("sizeof(int    ) = %d\n",sizeof(int    ));
    printf("sizeof(char  *) = %d\n",sizeof(char  *));
    printf("sizeof(char **) = %d\n",sizeof(char **));
    printf("//==============\n");
    test0();

    return 0;
}
```

编译：

gcc -m32 -o pointer_test pointer_test.c

运行：

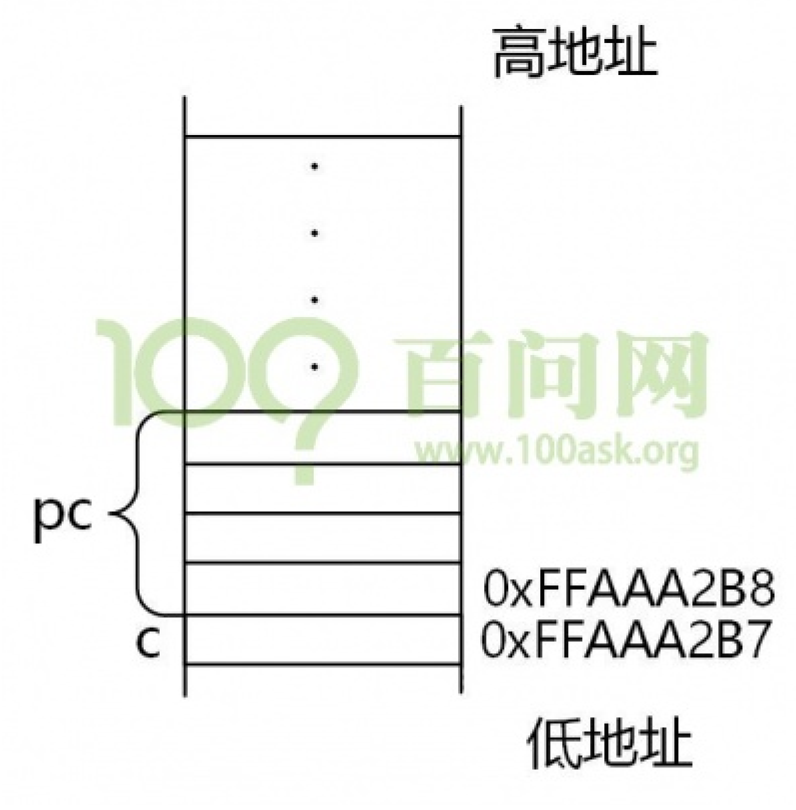
./pointer_test

结果：

```
sizeof(char    ) = 1
sizeof(int     ) = 4
sizeof(char   *) = 4
sizeof(char **) = 4
//==============
&c   =0xffaaa2b7
&pc =0xffaaa2b8
```

从运行的结果我们可知，变量 c 的地址编号(即地址)是 0xffaaa2b7，指针变量 pc 的地址编号是 0xffaaa2b8，如下图所示，编译成 32 位的机器码，字符类

型占用一个字节，指针类型就是用 4 个字节来存储的。



步骤二

我们把 test0()函数里面的变量保存(赋予)一些值，假如这些变量不保存数据的话，那么存储该变量的地址空间就会白白浪费，就相当于买个房子不住，就会白白浪费掉。

我们把上面程序中的 test0()函数里面的字符变量 c,指针变量 pc 进行赋值。

c = ‘A’; //把字符‘A’赋值给字符变量 c

pc = &c; //把字符变量 c 的地址赋值给指针变量 pc

然后把赋值后变量的值打印出来

```
printf("c   =%c\n",c);
printf("pc =%p\n",pc)
```

编译：

```
gcc -m32 -o pointer_test pointer_test.c
```

运行：

```
./pointer_test
```

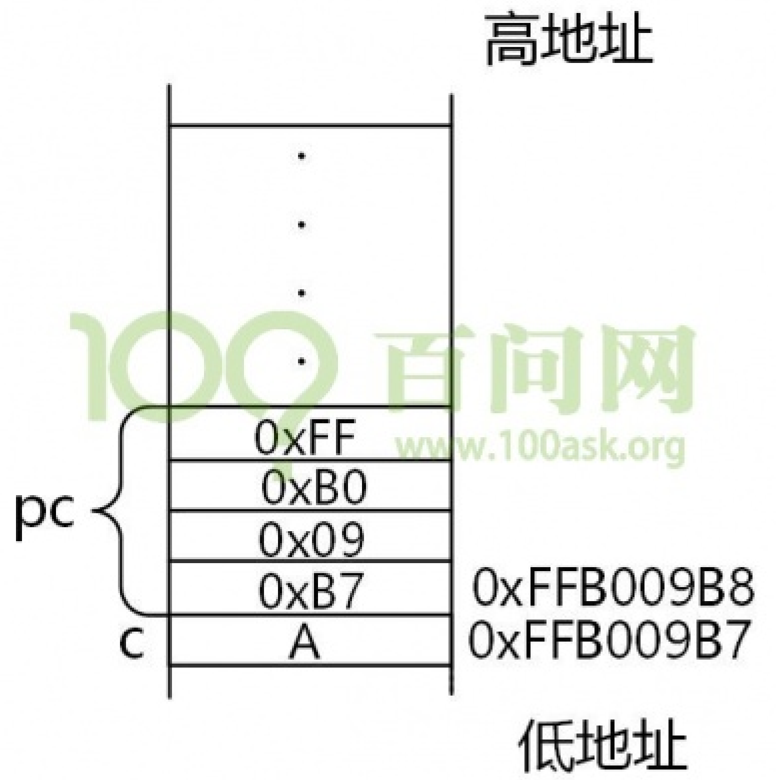
结果：

```
sizeof(char     ) = 1
sizeof(int      ) = 4
sizeof(char   *) = 4
sizeof(char **) = 4
//==============
&c    = 0xffb009b7
```

&pc = 0xffb009b8
c　=　A
pc = 0xffb009b7

从运行的结构来看字符变量和指针变量的地址编号发成了变化，所以在程序重新运行时，变量的地址，具有不确定性，字符变量 c 存储的内容是字符‘A’，指针变量 pc 存储的内容是 0xffb009b7（用四个字节来存储）。

由于内存的存储方式是，小端模式：低字节的数据放在低地址，高字节的数据放在高地址。在内存中的存储格式如下图所示。



步骤三

我们辛辛苦苦定义的指针类型变量，我们要把他用起来了，下面我们来分析一下，用指针来取值，‘*’：表示取指针变量存储地址的数据。

我们在 test0()函数里面添加如下代码：

```
printf("*pc =%c\n",*pc);
printf("//=================\n");
```

编译：

gcc -m32 -o pointer_test pointer_test.c

运行：

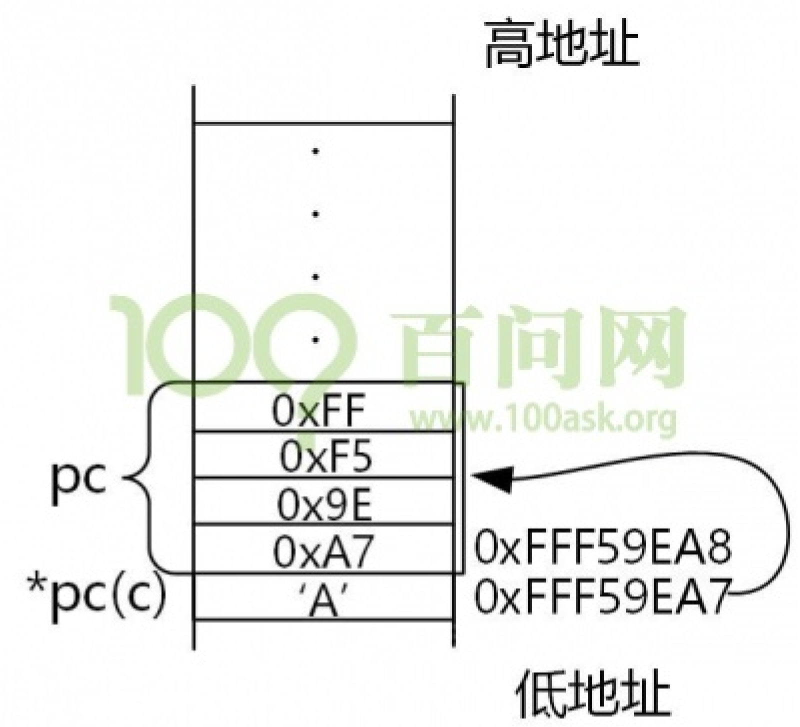
./pointer_test

结果：

```
sizeof(char     ) = 1
sizeof(int      ) = 4
sizeof(char    *) = 4
sizeof(char **) = 4
```

//==============
&c =0xfff59ea7
&pc =0xfff59ea8
c =A
pc =0xfff59ea7
*pc =A
//=================

指针变量 pc 存储的内容是是字符变量 c 的地址，所以*pc 就想相当于取字符变量 c 的内容。如



图

实例 1

步骤一

我们在上面函数的基础上，写一个函数 test1()

```
void test1()
{
    int  ia;
    int  *pi;
    char *pc;

    /* 第一步 ： 所有变量都保存在内存中，我们打印一下变量的存储地址
*/
    printf("&ia =%p\n",&ia);
    printf("&pi =%p\n",&pi);
    printf("&pc =%p\n",&pc);
```

```
}
```

main.c

```
int main(int argc, char *argv[])
{
  printf("sizeof(char   ) = %d\n",sizeof(char   ));
  printf("sizeof(int    ) = %d\n",sizeof(int    ));
  printf("sizeof(char  *) = %d\n",sizeof(char  *));
  printf("sizeof(char **) = %d\n",sizeof(char **));
  printf("//==============\n");
  //test0();
  test1();
  return 0;
}
```

我们在 test1()函数中定义了一个整型变量 ia，定义了一个指向整型的指针变量 pi，定义了一个指向字符型的指针变量 pc。然后打印出这些变量的地址。

编译

gcc -m32 -o pointer_test pointer_test.c

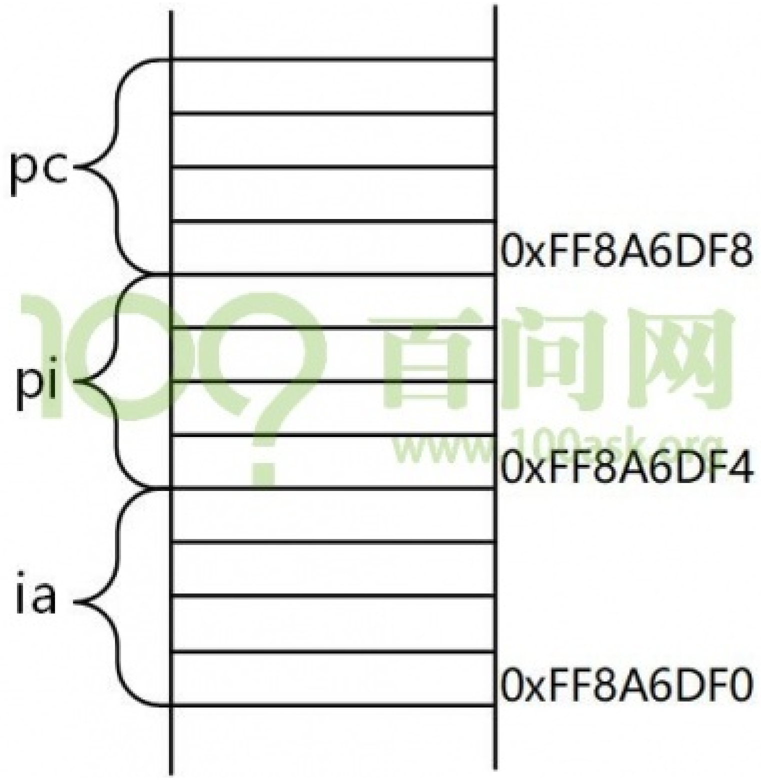
运行：

./pointer_test

结果：

```
sizeof(char    ) = 1
sizeof(int     ) = 4
sizeof(char   *) = 4
sizeof(char **) = 4
//==============
&ia =0xffc936e4
&pi =0xffc936e8
&pc =0xffc936ec
```

在 32 位的系统中 int 类型变量在内存中占用 4 个字节，指针型变量在内存中占用 4 个字节如图:



步骤二

在 test1()的函数中对定义的变量进行赋值，然后把赋值的结果打印出来。

```
/*第二步：所有变量都可以保存某些值，接着赋值并打印*/
    ia = 0x12345678;
    pi = &ia;
    pc = (char *)&ia;
    printf("ia =0x%x\n",ia);
    printf("pi =%p\n",pi);
    printf("pc =%p\n",pc);
```

编译

gcc -m32 -o pointer_test pointer_test.c

运行：

./pointer_test

结果：

```
sizeof(char   ) = 1
sizeof(int    ) = 4
sizeof(char  *) = 4
sizeof(char **) = 4
//==============
&ia = 0xffb6f724
&pi = 0xffb6f728
```

&pc = 0xffb6f72c
ia = 0x12345678
pi = 0xffb6f724
pc = 0xffb6f724

从结果可以看出来，变量 pi 和 pc 的值都等于变量 ia 的地址。

步骤三

我们使用指针并且对其进行取值，然后移动指针，在 test1 中添加如下代码，完成所述要求

```
/*第三步：使用指针：1)取值    2)移动指针*/
printf("*pi =0x%x\n",*pi); printf("pc =%p\t",pc); printf("*pc =0x%x\n",*pc);
pc=pc+1; printf("pc =%p\t",pc); printf("*pc =0x%x\n",*pc); pc=pc+1; printf("pc
=%p\t",pc); printf("*pc =0x%x\n",*pc); pc=pc+1; printf("pc =%p\t",pc); printf("*pc
=0x%x\n",*pc); printf("//=================\n");
```

编译

```
gcc -m32 -o pointer_test pointer_test.c
```

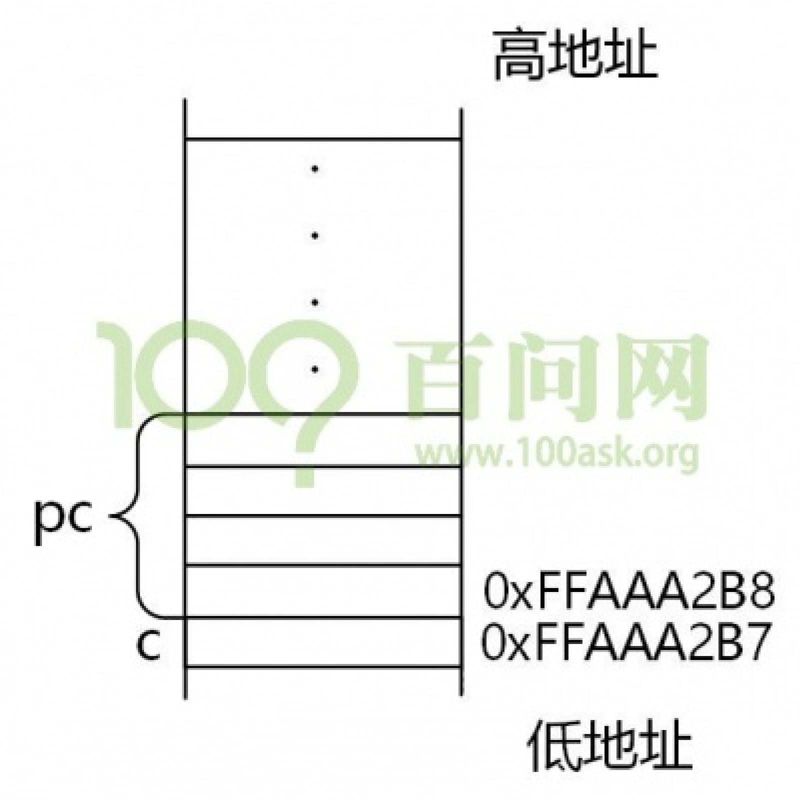
运行：

```
./pointer_test
```

结果：

```
sizeof(char      ) = 1
sizeof(int        ) = 4
sizeof(char    *) = 4
sizeof(char **) = 4
//==============
&ia =0xffee0930
&pi =0xffee0934
&pc =0xffee0938
ia =0x12345678
pi =0xffee0930
pc =0xffee0930
*pi =0x12345678
pc =0xffee0930    *pc =0x78
pc =0xffee0931    *pc =0x56
pc =0xffee0932    *pc =0x34
pc =0xffee0933    *pc =0x12
```

由于 pi 指向了 ia,所以*pi 的值为 0x12345678。由于 pc 也指向了 ia,但是由于 pc 是字符型指针变量，一次只能访问一个字节，需要四次才能访问完。如图所示：



结论：

1. 指针变量所存储的内容是所指向的变量在内存中的起始地址。

2. &变量：

目的：获得变量在内存中的地址；返回：变量在内存中起始地址；

第 004 节_c 语言指针复习 2_指向数组和字符串的指针

实例 2

我们在 pointer_test.c 的文件中写一个 test2()函数，我们定义一个有 3 个元素的字符数组初始化值分别为，’ A’，’ B’，’ C’，然后定义一个字符指针 pc，把数组 ca 的首地址复制给字符指针 pc,然后通过访问指针变量 pc,来读取指针变量 pc 所指向地址的数据，代码如下：

```
void test2()
{
   char ca[3]={'A','B','C'};
   char *pc;

   /*第一步 : 所有变量都保存在内存中，我们打印一下变量的存储地址
*/
   printf("ca  =%p\n",ca);
   printf("&pc =%p\n",&pc);

   /*第二步：所有变量都可以保存某些值，接着赋值并打印*/
```

```
    //前面已经有ca[3]={'A','B','C'};
    pc = ca;
    printf("pc =%p\n",pc);

    /*第三步：使用指针：1)取值  2)移动指针*/
    printf("pc =%p\t",pc); printf("*pc =0x%x\n",*pc);
pc=pc+1;
    printf("pc =%p\t",pc); printf("*pc =0x%x\n",*pc);
pc=pc+1;
    printf("pc =%p\t",pc); printf("*pc =0x%x\n",*pc);
    printf("//=================\n");
}
```

main()函数

```
int main(int argc,char **argv)
{
    printf("sizeof(char    )=%d\n",sizeof(char    ));
    printf("sizeof(int     )=%d\n",sizeof(int     ));
    printf("sizeof(char   *)=%d\n",sizeof(char   *));
    printf("sizeof(char **)=%d\n",sizeof(char **));
    printf("//=================\n");
    //test0();
    //test1();
    test2();
    return 0;
}
```

编译

gcc -m32 -o pointer_test pointer_test.c

运行：

./pointer_test

结果：

```
sizeof(char     ) = 1
sizeof(int      ) = 4
sizeof(char    *) = 4
sizeof(char **) = 4
//==============
ca   =0xffb946b9
&pc =0xffb946b4
pc =0xffb946b9
pc =0xffb946b9    *pc =0x41
pc =0xffb946ba    *pc =0x42
pc =0xffb946bb    *pc =0x43
//=================
```

分析：

第一步：

首先定义一个 3 个元素的字符数组 ca(数组名表示该数组存储的首地址)，然后定义一个字符指针 pc，然后通过 printf()函数把定义这两个变量在内存中的地址打印出来。

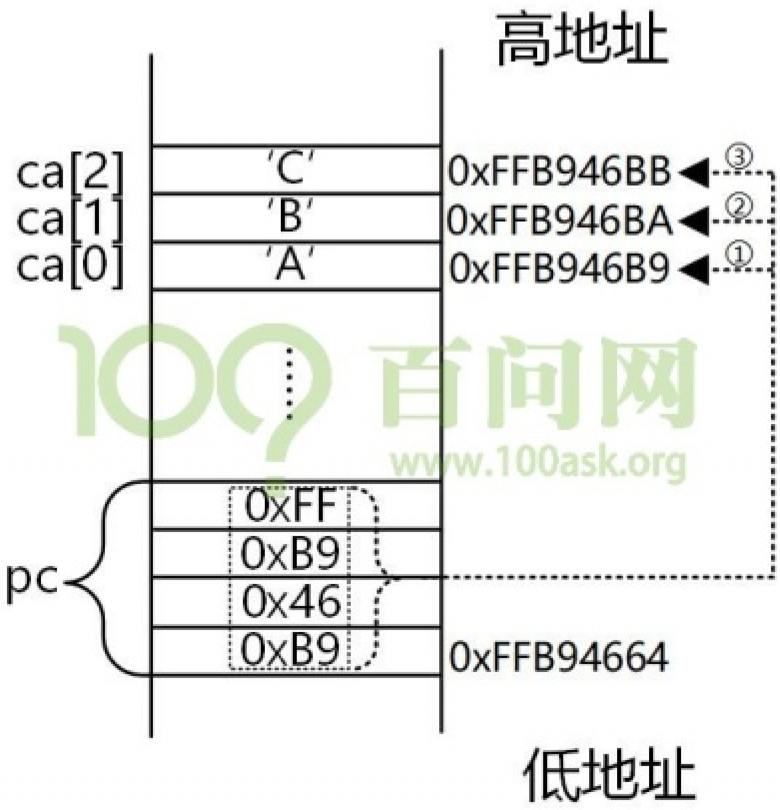
第二步：

执行 pc = ca;就是把数组 ca 的首地址复制给指针变量 pc，然后通过 printf()函数打印 pc 的值可以看出 pc 的值就是字符数组 ca 的首地址 0xffb946b9。

第三步：

通过移动指针我们可以发现数组所占用的内存是连续的，0x41(的 ascii 值‘A‘)，0x42(的 ascii 值‘B‘)，0x43(的 ascii 值‘C‘)。

如图



实例 3

我们在 pointer_test.c 的文件中写一个 test3()函数，我们定义一个有 3 个元素的整型数组 ia,初始化值分别为，0x12345678, 0x87654321, 0x13572468，然后定义一个整型指针 pi，把数组 ia 的首地址复制给整型指针 pi,然后通过访问指针变量 pi,来读取指针变量 pi 所指向地址的数据，代码如下：

```
void test3()
{
   int ia[3]={0x12345678,0x87654321,0x13572468};
```

```
    int *pi;

    /*第一步 : 所有变量都保存在内存中, 我们打印一下变量的存储地址
*/
    printf("ia  =%p\n",i);
    printf("&pi =%p\n",&pi);

    /*第二步: 所有变量都可以保存某些值,接着赋值并打印*/

    //前面已经有ia[3]={0x12345678,0x87654321,0x13572468};
    pi = ia;
    printf("pi =%p\n",pi);

    /*第三步: 使用指针: 1)取值  2)移动指针*/
    printf("pi =%p\t",pi); printf("*pi =0x%x\n",*pi);
pi=pi+1;
    printf("pi =%p\t",pi); printf("*pi =0x%x\n",*pi);
pi=pi+1;
    printf("pi =%p\t",pi); printf("*pi =0x%x\n",*pi);
    printf("//=================\n");
}
```

把 main()函数 test2()修改为 test3().

编译

gcc -m32 -o pointer_test pointer_test.c

运行:

./pointer_test

结果:

```
sizeof(char     ) = 1
sizeof(int      ) = 4
sizeof(char    *) = 4
sizeof(char **) = 4
//==============
ia   =0xff91c060
&pi =0xff91c05c
pi =0xff91c060
pi =0xff91c060    *pi =0x12345678
pi =0xff91c064    *pi =0x87654321
pi =0xff91c068    *pi =0x13572468
```

分析:

第一步:

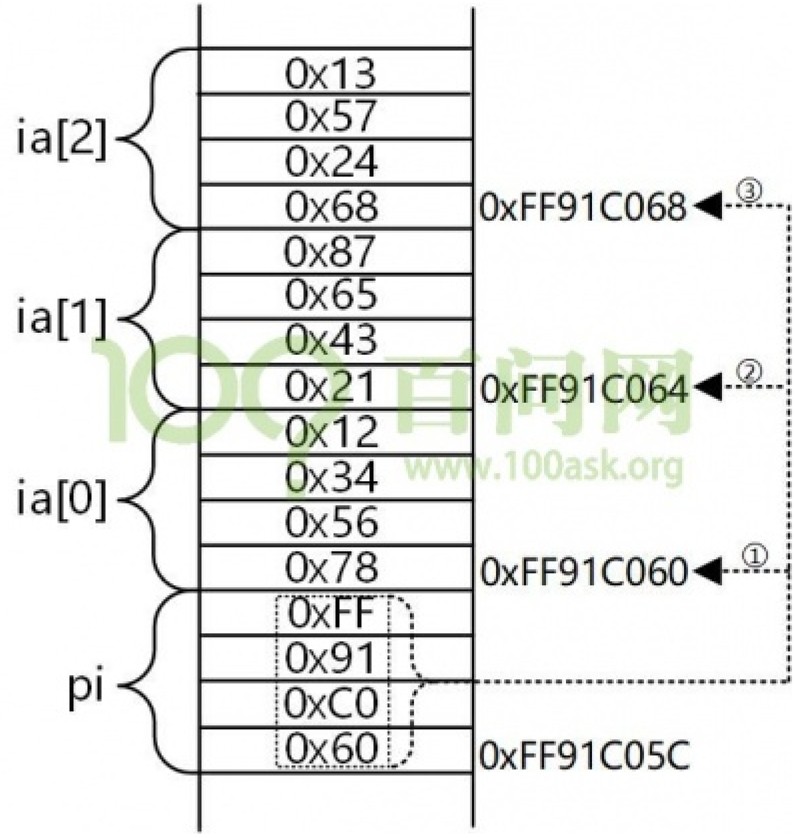
我们定义一个有 3 个元素的整型数组 ia 数组名表示该数组存储的首地址),初始化值分别为，0x12345678, 0x87654321, 0x13572468, 然后定义一个整型指针 pi，然后通过 printf()函数把定义这两个变量在内存中的地址打印出来。

第二步：

执行 pi = ia; 就是把数组 ia 的首地址复制给指针变量 pi，然后通过 printf()函数打印 pi 的值可以看出 pi 的值就是整型数组 ia 的首地址 0xff91c060。

第三步：

我们知道 pi 是整型指针变量，并且整型变量占用四个字节，所以整型指针变量 pi 是以四字节为单元进行访问的，所以 pi 和 pi+1 之间的差是一个整型变量的大小（4 个字节）。



实例 4

定义一个指向字符串的指针 pc，然后对字符串指针进行初始化设置为 **abc**，代码如下：

```
void test4()
{
   char *pc="abc";
   /*第一步 ： 所有变量都保存在内存中，我们打印一下变量的存储地址
*/
   printf("&pc =%p\n",&pc);
```

```
    /*第二步：所有变量都可以保存某些值，接着赋值并打印*/
    //前面已经有pc="abc";

    /*第三步：使用指针：1)取值  2)移动指针*/
    printf("pc    =%p\n", pc);
    printf("*pc   =%c\n",*pc);
    printf("pc str=%s\n", pc);
}
```

把 main()函数 test3()修改为 test4(). 编译

gcc -m32 -o pointer_test pointer_test.c

运行：

./pointer_test

结果：

```
sizeof(char    ) = 1
sizeof(int     ) = 4
sizeof(char   *) = 4
sizeof(char **) = 4
//==============
&pc    =0xfff49a68
pc     =0x08048b4b
*pc    =a
pc str=abc
```

分析：

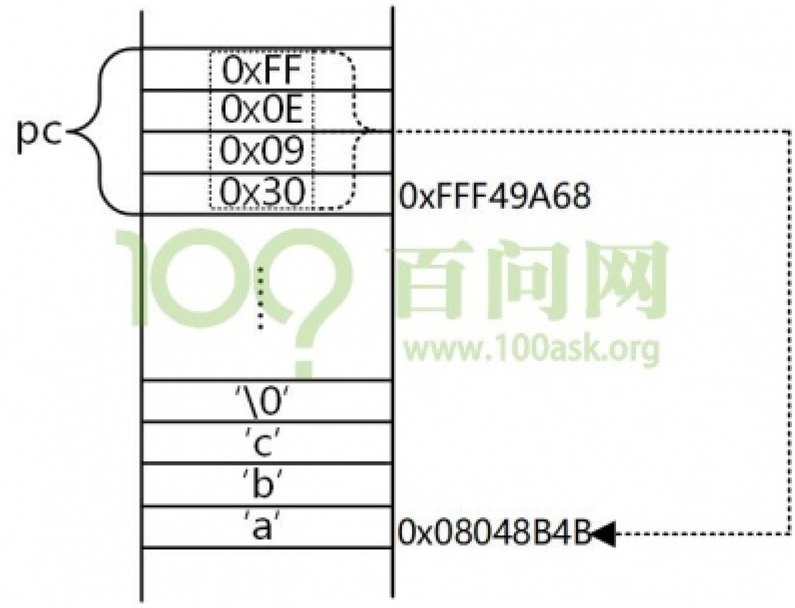
第一步：

定义一个指向字符串的指针 pc，然后对字符串指针进行初始化设置为 **abc**，此时，指针变量 pc 的值就是字符串 **abc** 的首地址，然后通过 printf()函数把指针 pc 的地址打印出来为 0xfff49a68

第二步：

首先通过 printf()函数打印出指针变量 pc 的值(字符串 **abc** 的首地址)，pc 的值为 0x08048b4b，然后通过 pc 指针访问第一个字符(pc 的就是字符串的首地

址)，所以 pc 的值就是字符‘a‘的地址，所以*pc 的值就是’a‘， 如图所示:



下面分析一下指向数组的指针和指向字符串的指针:

char ca[3]={'A','B','C'};

char *pc0 = ca;

pc0 是指向字符数组的字符指针,pc0 就是数组首元素的地址,pc0=&a[0]

char *pc11="abc";

pc 是指向字符串的字符指针,pc1 就是字符串"abc"的首字符'a'的地址。

第 005 节_Makefile 的引入及规则

使用 keil, mdk, avr 等工具开发程序时点击鼠标就可以编译了，它的内部机制是什么？它怎么组织管理程序？怎么决定编译哪一个文件？

答：实际上 windows 工具管理程序的内部机制，也是 Makefile，我们在 linux 下来开发裸板程序的时候，使用 Makefile 组织管理这些程序，本节我们来讲解 Makefile 最基本的规则。Makefile 要做什么事情呢？ 组织管理程序，组织管理文件，我们写一个程序来实验一下：

文件 a.c

```
02 #include <stdio.h>
03
04 int main()
05 {
06 func_b();
07 return 0;
08}
```

文件 b.c

```
2  #include <stdio.h>
```

```
3
4  void func_b()
5  {
6          printf("This is B\n");
7  }
```

编译：

gcc -o test a.c b.c

运行：

./test

结果：

This is B

`gcc -o test a.c b.c`这条命令虽然简单，但是它完成的功能不简单。 我们来看看它做了哪些事情。

我们知道.c 程序 --> 得到可执行程序

它们之间要经过四个步骤：

1. 预处理
2. 编译
3. 汇编
4. 链接

我们经常把前三个步骤统称为编译了。我们具体分析：`gcc -o test a.c b.c`这条命令 它们要经过下面几个步骤：

1）. 对于 a.c 执行：预处理 编译 汇编 的过程，a.c -->xxx.s -->xxx.o 文件。

2）. 对于 b.c 执行：预处理 编译 汇编 的过程，b.c -->yyy.s -->yyy.o 文件。

3）. 最后：xxx.o 和 yyy.o 链接在一起得到一个 test 应用程序。

提示：`gcc -o test a.c b.c -v` ：加上一个‘-v’选项可以看到它们的处理过程，

第一次编译 a.c 得到 xxx.o 文件，这是很合乎情理的， 执行完第一次之后，如果修改 a.c 又再次执行：`gcc -o test a.c b.c`，对于 a.c 应该重新生成 xxx.o，但是对于 b.c 又会重新编译一次，这完全没有必要，b.c 根本没有修改，直接使用第一次生成的 yyy.o 文件就可以了。

缺点：对所有的文件都会再处理一次，即使 b.c 没有经过修改，b.c 也会重新编译一次， 当文件比较少时，这没有没有什么问题，当文件非常多的时候，就会带来非常多的效率问题。

如果文件非常多的时候，我们，只是修改了一个文件，所用的文件就会重新处理一次，编译的时候就会等待很长时间。

对于这些源文件，我们应该分别处理，执行：预处理 编译 汇编 ，先分别编译它们，最后再把它们链接在一次，比如：

编译：

```
gcc -o a.o a.c
gcc -o b.o b.c
```

链接：

```
gcc -o test a.o b.o
```

比如：上面的例子，当我们修改 a.c 之后,a.c 会重现编译然后再把它们链接在一起就可以了。，b.c 就不需要重新编译。

那么问题又来了，怎么知道哪些文件被更新了/被修改了？

比较时间：比较 a.o 和 a.c 的时间，如果 a.c 的时间比 a.o 的时间更加新的话，就表明 a.c 被修改了，同理 b.o 和 b.c 也会进行同样的比较。比较 test 和 a.o, b.o 的时间，如果 a.o 或者 b.o 的时间比 test 更加新的话，就表明应该重新生成 test。Makefile 就是这样做的。

我们现在来写出一个简单的 Makefile：makefie 最基本的语法是规则，规则：

```
目标  :   依赖 1   依赖 2  ...
[TAB]命令
```

当“依赖”比“目标”新，执行它们下面的命令。我们要把上面三个命令写成 makefile 规则，如下：

test ：a.o b.o //test 是目标，它依赖于 a.o b.o 文件，一旦 a.o 或者 b.o 比 test 新的时候，

就需要执行下面的命令，重新生成 test 可执行程序。

```
gcc -o test a.o b.o
```

a.o : a.c *//a.o 依赖于 a.c，当 a.c 更加新的话，执行下面的命令来生成 a.o*

```
gcc -c -o a.o a.c
```

```
b.o : b.c  //b.o 依赖于 b.c,当 b.c 更加新的话，执行下面的命令，来生成 b.o
    gcc -c -o b.o b.c
```

我们来作一下实验：
在改目录下我们写一个 Makefile 文件：
文件：Makefile

```
1  test:a.o b.o
2          gcc -o test a.o b.o
3
4  a.o : a.c
5          gcc -c -o a.o a.c
6
7  b.o : b.c
8          gcc -c -o b.o b.c
```

上面是 makefile 中的三条规则。makefile,就是名字为“makefile”的文件。当我们想编译程序时，直接执行 make 命令就可以了，一执行 make 命令它想生成第一个目标 test 可执行程序，如果发现 a.o 或者 b.o 没有，就要先生成 a.o 或者 b.o，发现 a.o 依赖 a.c，有 a.c

但是没有 a.o,他就会认为 a.c 比 a.o 新，就会执行它们下面的命令来生成 a.o，同理 b.o 和 b.c 的处理关系也是这样的。

如果修改 a.c ，我们再次执行 make，它的本意是想生成第一个目标 test 应用程序，

它需要先生成 a.o，发现 a.o 依赖 a.c(执行我们修改了 a.c)发现 a.c 比 a.o 更加新，就会执行 `gcc -c -o a.o a.c` 命令来生成 a.o 文件。b.o 依赖 b.c，发现 b.c 并没有修改，就不会执行 `gcc -c -o b.o b.c` 来重新生成 b.o 文件。现在 a.o b.o 都有了，其中的 a.o 比 test 更加新，就会执行 `gcc -o test a.o b.o` 来重新链接得到 test 可执行程序。所以当执行 make 命令时候就会执行下面两条执行：

```
gcc -c -o a.o a.c
gcc -o test a.o b.o
```

我们第一次执行 make 的时候，会执行下面三条命令(三条命令都执行)：
gcc -c -o a.o a.c gcc -c -o b.o b.c gcc -o test a.o b.o
</syntaxhighlight>

再次执行 make 就会显示下面的提示：

```
make: `test' is up to date.
```

我们再次执行 make 就会判断 Makefile 文件中的依赖，发现依赖没有更新，所以目标文件就不会重现生成，就会有上面的提示。当我们修改 a.c 后，重新执行 make,
就会执行下面两条指令：

```
gcc -c -o a.o a.c
gcc -o test a.o b.o
```

我们同时修改 a.c b.c，执行 make 就会执行下面三条指令。

```
gcc -c -o a.o a.c
gcc -c -o b.o b.c
gcc -o test a.o b.o
```

a.c 文件修改了，重新编译生成 a.o，b.c 修改了重新编译生成 b.o，a.o, b.o 都更新了重新链接生成 test 可执行程序，makefile 的规则其实还是比较简单的。

规则是 Makefie 的核心，执行 make 命令的时候，就会在当前目录下面找到名字为：Makefile 的文件，根据里面的内容来执行里面的判断/命令。

第 006 节_Makefile 的语法

本节我们只是简单的讲解 Makefile 的语法，如果想比较深入学习 Makefile 的话可以：

a. 百度搜 "gnu make 于凤昌"。

b. 查看官方文档: http://www.gnu.org/software/make/manual/

通配符

假如一个目标文件所依赖的依赖文件很多，那样岂不是我们要写很多规则，这显然是不合乎常理的。

我们可以使用通配符，来解决这些问题。

我们对上节程序进行修改代码如下：

```
test: a.o b.o
	gcc -o test $^

%.o : %.c
	gcc -c -o $@ $<
```

%.o：表示所用的.o 文件

%.c：表示所有的.c 文件

$@：表示目标

$<：表示第 1 个依赖文件

$^：表示所有依赖文件

我们来在该目录下增加一个 c.c 文件，代码如下：

```
#include <stdio.h>
```

```
void func_c()
{
    printf("This is C\n");
}
```

然后在 main 函数中调用修改 Makefile，修改后的代码如下：

```
test: a.o b.o c.o
    gcc -o test $^

%.o : %.c
    gcc -c -o $@ $<
```

执行：

```
make
```

结果：

```
gcc -c -o a.o a.c
gcc -c -o b.o b.c
gcc -c -o c.o c.c
gcc -o test a.o b.o c.o
```

运行：

```
./test
```

结果：

```
This is B
This is C
```

假想目标: .PHONY

1.我们想清除文件，我们在 Makefile 的结尾添加如下代码就可以了：

```
clean:
    rm *.o test
```

1）.执行 make：生成第一个可执行文件。

2）.执行 make clean: 清除所有文件，即执行：rm *.o test。

make 后面可以带上目标名，也可以不带，如果不带目标名的话它就想生成第一个规则里面的第一个目标。

2.使用 Makefile

执行：make [目标]

也可以不跟目标名，若无目标默认第一个目标。我们直接执行 make 的时候，会在 makefile 里面找到第一个目标然后执行下面的指令生成第一个目标。当我们执行 make clean 的时候，就会在 Makefile 里面找到 clean 这个目标，然后执行里面的命令，这个写法有些问题，原因是我们的目录里面没有 clean 这个文件，这个规则执行的条件成立，他就会执行下面的命令来删除文件。

如果：该目录下面有名为 clean 文件怎么办呢？

我们在该目录下创建一个名为“clean”的文件，然后重新执行：make 然后 make clean，结果(会有下面的提示：)：

```
make: `clean' is up to date.
```

它根本没有执行我们的删除操作，这是为什么呢？

我们之前说，一个规则能过执行的条件：1),目标文件不错在 2),依赖文件比目标新。现在我们的目录里面有名为“clean”的文件，目标文件是有的，并且没有依赖文件，没有办法判断依赖文件的时间。这种写法会导致：有同名的"clean"文件时，就没有办法执行 make clean 操作。解决办法：我们需要把目标定义为假象目标，用关键字 PHONY。

.PHONY: clean //把 clean 定义为假象目标。他就不会判断名为“clean”的文件是否存在，

然后在 Makfile 结尾添加.PHONY: clean 语句，重新执行：make clean，就会执行删除操作。

变量

在 makefile 中有两种变量：

1)简单变量(即使变量)：

A := xxx # A 的值即刻确定，在定义时即确定

对于即使变量使用“:=”表示，它的值在定义的时候已经被确定了

2）延时变量

B = xxx # B 的值使用到时才确定

对于延时变量使用“=”表示。它只有在使用到的时候才确定，在定义/等于时并没有 确定下来。

想使用变量的时候使用“$”来引用，如果不想看到命令是，可以在命令的前面加上"@"符号，就不会显示命令本身。当我们执行 make 命令的时候，make 这个指令本身，会把整个 Makefile 读进去，进行全部分析，然后解析里面的变量。常用的变量的定义如下：

:= # 即时变量

= # 延时变量

?= # 延时变量, 如果是第 1 次定义才起效, 如果在前面该变量已定义则忽略这句

+= # 附加, 它是即时变量还是延时变量取决于前面的定义

?=：如果这个变量在前面已经被定义了，这句话就会不会起效果，

实例：

```
A := $(C)
B = $(C)
C = abc

#D = 100ask
D ?= weidongshan

all:
	@echo A = $(A)
	@echo B = $(B)
	@echo D = $(D)

C += 123
```

执行：

```
make
```

结果：

```
A =
B = abc 123
D = weidongshan
```

分析：

1) A := $(C)：　A 为即使变量，在定义时即确定，由于刚开始 C 的值为空，所以 A 的值也为空。

2) B = $(C)：　B 为延时变量，只有使用到时它的值才确定，当执行 make 时，会解析 Makefile 里面的所用变量，所以先解析 C = abc,然后解析 C += 123，此时，C = abc 123，当执行：@echo B = $(B) B 的值为 abc 123。

3) D ?= weidongshan：　D 变量在前面没有定义，所以 D 的值为 weidongshan，如果在前面添加 D = 100ask，最后 D 的值为 100ask。

我们还可以通过命令行存入变量的值 例如：

执行：

```
make D=123456
```

里面的 D ?= weidongshan 这句话就不起作用了。

结果：

```
A =
B = abc 123
D = 123456
```

第 007 节_Makefile 函数

makefile 里面可以包含很多函数，这些函数都是 make 本身实现的，下面我们来几个常用的函数。

引用一个函数用“$”。

函数 foreach

函数 foreach 语法如下：

$(foreach var,list,text)

前两个参数，‘var’和‘list’，将首先扩展，注意最后一个参数‘text’此时不扩展；接着，对每一个‘list’扩展产生的字，将用来为‘var’扩展后命名的变量赋值；然后‘text’引用该变量扩展；因此它每次扩展都不相同。结果是由空格隔开的‘text’ 在‘list’中多次扩展的字组成的新的‘list’。‘text’多次扩展的字串联起来，字与字之间由空格隔开，如此就产生了函数 foreach 的返回值。

实例：

```
A = a b c
B = $(foreach f, &(A), $(f).o)

all:
	@echo B = $(B)
```

结果：

```
B = a.o b.o c.o
```

函数 filter/filter-out

函数 filter/filter-out 语法如下：

$(filter pattern...,text)　　# 在 text 中取出符合 patten 格式的值

$(filter-out pattern...,text)　# 在 text 中取出不符合 patten 格式的值

实例：

```
C = a b c d/

D = $(filter %/, $(C))
E = $(filter-out %/, $(C))

all:
	@echo D = $(D)
	@echo E = $(E)
```

结果：

```
D = d/
E = a b c
```

Wildcard

函数 Wildcard 语法如下：

$(wildcard pattern)　# pattern 定义了文件名的格式, wildcard 取出其中存在的文件

这个函数 wildcard 会以 pattern 这个格式，去寻找存在的文件，返回存在文件的名字。

实例：

在该目录下创建三个文件：a.c b.c c.c

```
files = $(wildcard *.c)

all:
	@echo files = $(files)
```

结果：

```
files = a.c b.c c.c
```

我们也可以用 wildcard 函数来判断，真实存在的文件 实例：

```
files2 = a.c b.c c.c d.c e.c  abc
files3 = $(wildcard $(files2))

all:
	@echo files3 = $(files3)
```

结果：

```
files3 = a.c b.c c.c
```

patsubst 函数

函数 patsubst 语法如下：

$(patsubst pattern,replacement,$(var))

patsubst 函数是从 var 变量里面取出每一个值，如果这个符合 pattern 格式，把它替换成 replacement 格式。

实例：

```
files2  = a.c b.c c.c d.c e.c abc

dep_files = $(patsubst %.c,%.d,$(files2))

all:
	@echo dep_files = $(dep_files)
```

结果：

```
dep_files = a.d b.d c.d d.d e.d abc
```

第 008 节_Makefile 实例

前面讲了那么多 Makefile 的知识，现在开始做一个实例。

之前编译的程序 002_syntax，有个缺陷，将其复制出来，新建一个 003_example 文件夹，放在里面。 在 c.c 里面，包含一个头文件 c.h，在 c.h 里面定义一个宏，把这个宏打印出来。

c.c:

```
#include <stdio.h>
#include <c.h>

void func_c()
{
   printf("This is C = %d\n", C);
}
```

c.h:

```
#define C 1
```

然后上传编译，执行./test,打印出：

```
This is B
This is C =1
```

测试没有问题，然后修改 c.h：

```
#define C 2
```

重新编译，发现没有更新程序，运行，结果不变，说明现在的 Makefile 存在问题。

为什么会出现这个问题呢， 首先我们 test 依赖 c.o，c.o 依赖 c.c，如果我们更新 c.c，会重新更新整个程序。 但 c.o 也依赖 c.h，我们更新了 c.h，并没有在 Makefile 上体现出来，导致 c.h 的更新，Makefile 无法检测到。 因此需要添加:

```
c.o : c.c c.h
```

现在每次修改 c.h，Makefile 都能识别到更新操作，从而更新最后输出文件。

这样又冒出了一个新的问题，我们怎么为每个.c 文件添加.h 文件呢？对于内核，有几万个文件，不可能为每个文件依次写出其头文件。 因此需要做出改

进，让其自动生成头文件依赖，可以参考这篇文章：
http://blog.csdn.net/qq1452008/article/details/50855810

```
gcc -M c.c // 打印出依赖

gcc -M -MF c.d c.c  // 把依赖写入文件c.d

gcc -c -o c.o c.c -MD -MF c.d  // 编译c.o, 把依赖写入文件c.d
```

修改 Makefile 如下：

```
objs = a.o b.o c.o

dep_files := $(patsubst %,.%.d, $(objs))
dep_files := $(wildcard $(dep_files))

test: $(objs)
	gcc -o test $^

ifneq ($(dep_files),)
include $(dep_files)
endif

%.o : %.c
	gcc -c -o $@ $< -MD -MF .$@.d

clean:
	rm *.o test

distclean:
	rm $(dep_files)

.PHONY: clean
```

首先用 obj 变量将.o 文件放在一块。 利用前面讲到的函数，把 obj 里所有文件都变为.%.d 格式，并用变量 dep_files 表示。 利用前面介绍的 wildcard 函数，判断 dep_files 是否存在。 然后是目标文件 test 依赖所有的.o 文件。 如果 dep_files 变量不为空，就将其包含进来。 然后就是所有的.o 文件都依赖.c 文件，且通过-MD -MF 生成.d 依赖文件。 清理所有的.o 文件和目标文件 清理依赖.d 文件。

现在我们修改了任何.h 文件，最终都会影响最后生成的文件，也没任何手工添加.h、.c、.o 文件，完成了支持头文件依赖。

下面再添加 CFLAGS，即编译参数。比如加上编译参数-Werror，把所有的警告当成错误。

```
CFLAGS = -Werror -Iinclude
```

…………

```
%.o : %.c
    gcc $(CFLAGS) -c -o $@ $< -MD -MF .$@.d
```

现在重新 make，发现以前的警告就变成了错误，必须要解决这些错误编译才能进行。在 a.c 里面声明一下函数：

```
void func_b();
void func_c();
```

重新 make，错误就没有了。

除了编译参数-Werror，还可以加上-I 参数，指定头文件路径，-Iinclude 表示当前的 inclue 文件夹下。 此时就可以把 c.c 文件里的#include ".h"改为 #include <c.h>，前者表示当前目录，后者表示编译器指定的路径和 GCC 路径。

第 010 课 掌握 ARM 芯片时钟体系

第 001 节_S3C2440 时钟体系结构

S3C2440 是 System On Chip(SOC)，在芯片上不仅仅有 CPU 还有一堆外设。至于有哪些外设，可以查看参考手册。在 S3C2440 参考手册的第 PRODUCT OVERVIEW 里面有个 BLOCK DIAGRAM 图：

BLOCK DIAGRAM

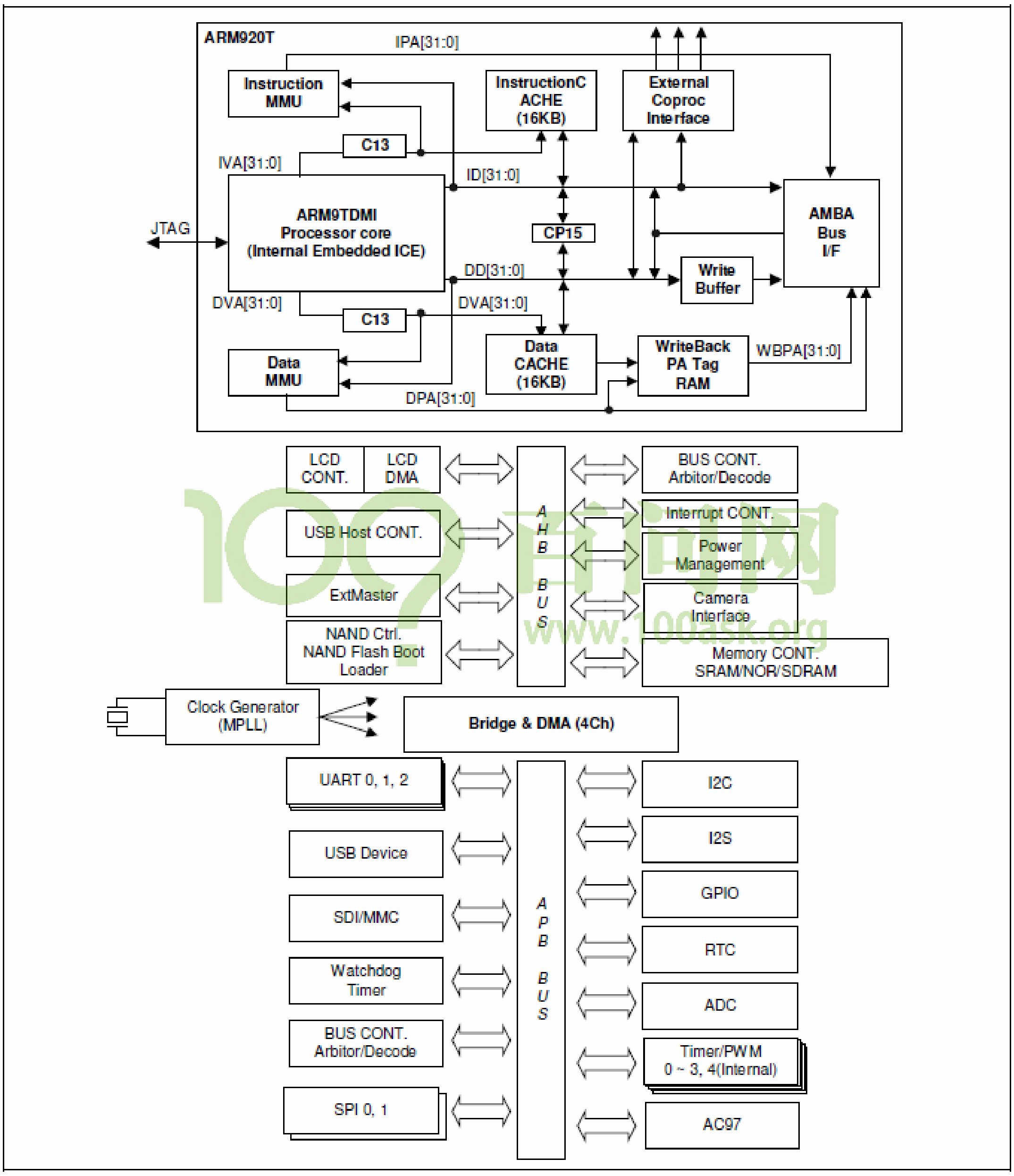


Figure 1-1. S3C2440A Block Diagram

可以把该图分为上中下三块，上面的是与 CPU 密切相关的，工作于 FCLK；中间的一些对性能要求较高的设备，像 LCD 显示、相机等，在 AHB BUS，H 即为 High，高速之意，工作于 HCLK；下面的是一些对性能要求不那么高的低速设备，在 APB BUS，P 即为 Peripheral 之意，工作在 PCLK。

在参考手册的特性里介绍了 S3C2440 的工作频率，Fclk 最高 400MHz，Hclk 最高 136MHz，Pclk 最高 68MHz。

如何得到以上的三种时钟？

硬件电路上有个 12M 的晶振，作为时钟源产生 12MHz 的频率，经过 SOC 的 PLL(锁相环)倍频产生 Fclk、Hclk、Pclk。

再具体看看第 7 章的时钟，在 Clock Generator Block Diagram 展示了时钟的产生。

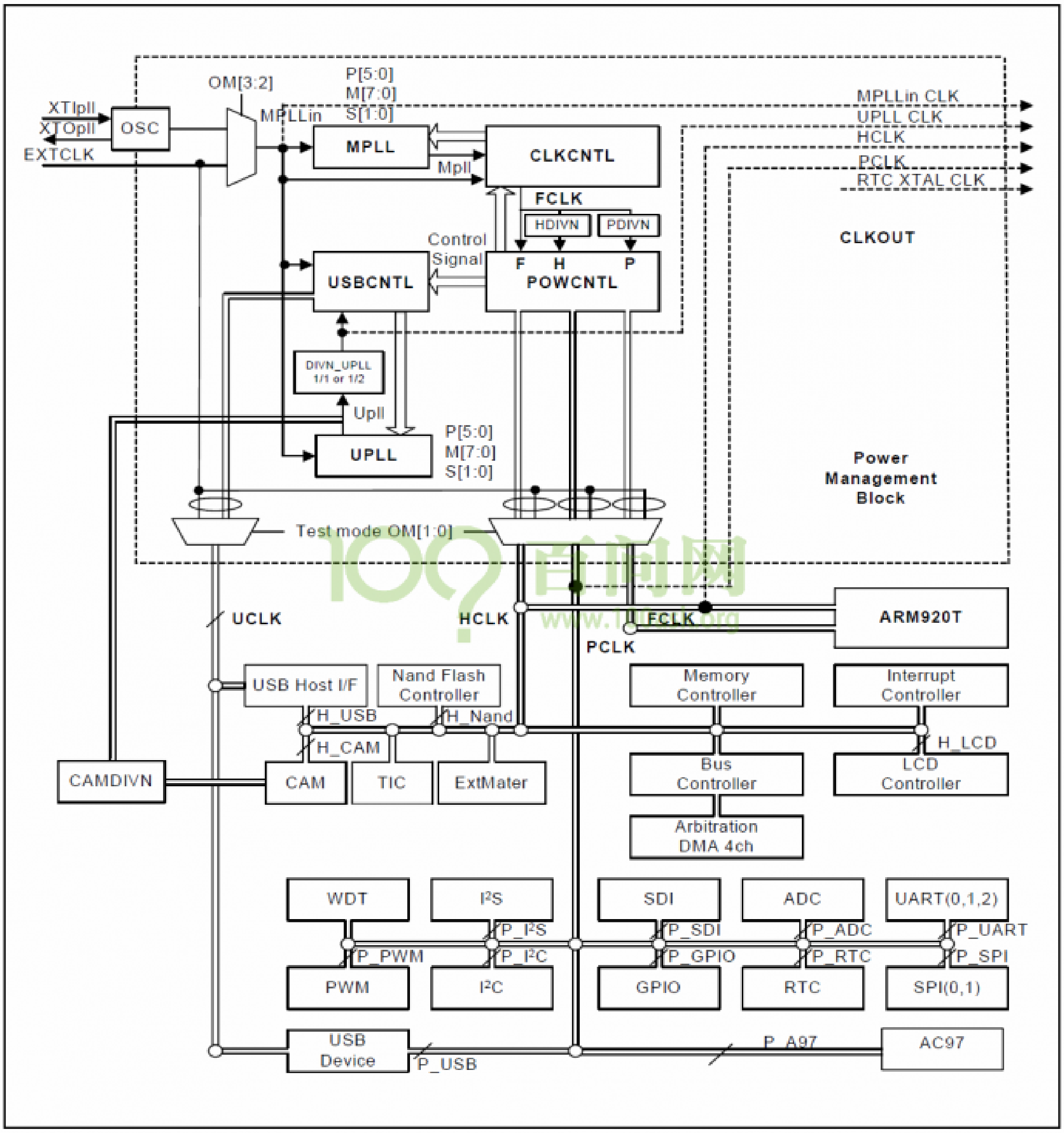
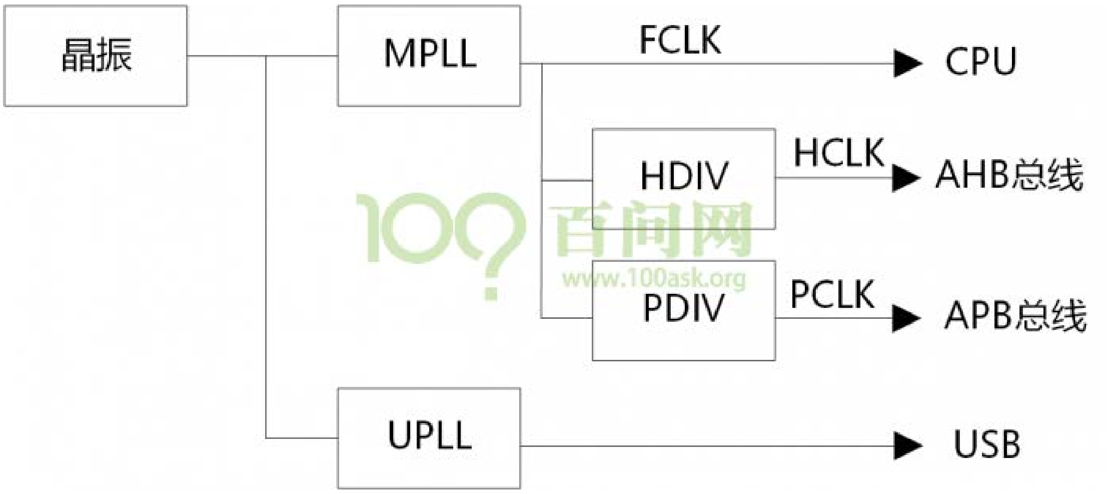


Figure 7-1. Clock Generator Block Diagram

在该图的左上角，晶振和一个外部时钟接在一个选择器上，这个选择器通过 OM[3:2]的值来决定选择哪个时钟源。然后生成的 MPLL(Main PLL)和 UPLL(USB PLL)，MPLL 直接提供给 FCLK，通过 HDIVN 分频给 HCLK，通过 PDIVN

分频给 PCLK，再传给下面的各个设备。



第 002 节_编程提高运行时钟

怎么编程控制 MPLL、HDIV、PDIV，使 FCLK=400MHz，HCLK=100MHz，PLCK=50MHz？

需要设置 MPLLCON 的 FCLK=400MHz，设置 CLKDIVN 的 HCLK=FCLK/4，PCLK=FCLK/8。

1. 首先看 CLKDIVN 寄存器：

Register	Address	R/W	Description	Reset Value
CLKDIVN	0x4C000014	R/W	Clock divider control register	0x00000000

CLKDIVN	Bit	Description	Initial State
DIVN_UPLL	[3]	UCLK select register(UCLK must be 48MHz for USB) 0: UCLK = UPLL clock 1: UCLK = UPLL clock / 2 Set to 0, when UPLL clock is set as 48MHz Set to 1. when UPLL clock is set as 96MHz.	0
HDIVN	[2:1]	00 : HCLK = FCLK/1. 01 : HCLK = FCLK/2. 10 : HCLK = FCLK/4 when CAMDIVN[9] = 0. HCLK= FCLK/8 when CAMDIVN[9] = 1. 11 : HCLK = FCLK/3 when CAMDIVN[8] = 0. HCLK = FCLK/6 when CAMDIVN[8] = 1.	00
PDIVN	[0]	0: PCLK has the clock same as the HCLK/1. 1: PCLK has the clock same as the HCLK/2.	0

想设置 HCLK=FCLK/4 需要将 HDIVN[2:1]设置为 10，同时将 CAMDIVN[9]设置为 0。

查看 CAMDIVN[9]的初始值默认就是 0，因此只需要设置 HDIVN[2:1]为 10。

CAMDIVN	Bit	Description	Initial State
DVS_EN	[12]	0:DVS OFF ARM core will run normally with FCLK (MPLLout). 1:DVS ON ARM core will run at the same clock as system clock (HCLK).	0
Reserved	[11]	–	0
Reserved	[10]	–	0
HCLK4_HALF	[9]	HDIVN division rate change bit, when CLKDIVN[2:1]=10b. 0: HCLK = FCLK/4 1: HCLK= FCLK/8 Refer the CLKDIV register.	0

想设置 PCLK=FCLK/8 需要将 PDIVN[0]设置为 1，因此整个 CLKDIVN 寄存器设置如下：

```
    /* CLKDIVN(0x4C000014) = 0X5, tFCLK:tHCLK:tPCLK = 1:4:8  */
        ldr r0, =0x4C000014
        ldr r1, =0x5
        str r1, [r0]
```

2. 现在看如何使 FCLK=400MHz.

在手册的 PLL VALUE SELECTION TABLE 里列出了常见情况 PLL 的设置，我们输入的是晶振的 12MHz，输出需要 400MHz，因此根据表格需要设置 MDIV=92(0x5C)，PDIV=1，SDIV=1；

Input Frequency	Output Frequency	MDIV	PDIV	SDIV
12.0000MHz	48.00 MHz (Note)	56(0x38)	2	2
12.0000MHz	96.00 MHz (Note)	56(0x38)	2	1
12.0000MHz	271.50 MHz	173(0xad)	2	2
12.0000MHz	304.00 MHz	68(0x44)	1	1
12.0000MHz	400.00 MHz	92(0x5c)	1	1

在手册介绍了 MPLL 的 m、p、s 与 MDIV、PDIV、SDIV 之间的关系：

```
Mpll = (2 * m * Fin) / (p * 2^S)
m = (MDIV + 8), p = (PDIV + 2), s = SDIV

m=MDIV+8=92+8=100
p=PDIV+2=3
s=SDIV=1
MPLL=2x100x12/(3x2^1)=400MHz
```

PLL 控制寄存器如下：

PLL CONTROL REGISTER (MPLLCON & UPLLCON)

Register	Address	R/W	Description	Reset Value
MPLLCON	0x4C000004	R/W	MPLL configuration register	0x00096030
UPLLCON	0x4C000008	R/W	UPLL configuration register	0x0004d030

PLLCON	Bit	Description	Initial State
MDIV	[19:12]	Main divider control	0x96 / 0x4d
PDIV	[9:4]	Pre-divider control	0x03 / 0x03
SDIV	[1:0]	Post divider control	0x0 / 0x0

因此需要配置（92<<12）|(1<<4)|(1<<0),

```
/* 设置 MPLLCON(0x4C000004) = (92<<12)|(1<<4)|(1<<0)
     *  m = MDIV+8 = 92+8=100
     *  p = PDIV+2 = 1+2 = 3
     *  s = SDIV = 1
     *  FCLK = 2*m*Fin/(p*2^s) = 2*100*12/(3*2^1)=400M
     */
    ldr r0, =0x4C000004
    ldr r1, =(92<<12)|(1<<4)|(1<<0)
    str r1, [r0]
```

3.此外，手册还提到，需要 SetAsyncBusMode。

MMU_SetAsyncBusMode

mrc p15,0,r0,c1,c0,0

orr r0,r0,#R1_nF:OR:R1_iA

mcr p15,0,r0,c1,c0,0

完整的 start.S:

```
.text
.global _start

_start:

    /* 关闭看门狗 */
    ldr r0, =0x53000000
    ldr r1, =0
    str r1, [r0]


    /* 设置 MPLL, FCLK : HCLK : PCLK = 400m : 100m : 50m
*/
    /* LOCKTIME(0x4C000000) = 0xFFFFFFFF */
    ldr r0, =0x4C000000
```

```
    ldr r1, =0xFFFFFFFF
    str r1, [r0]

    /* CLKDIVN(0x4C000014) = 0X5, tFCLK:tHCLK:tPCLK =
1:4:8  */
    ldr r0, =0x4C000014
    ldr r1, =0x5
    str r1, [r0]


    /* 设置CPU工作于异步模式 */

    mrc p15,0,r0,c1,c0,0
    orr r0,r0,#0xc0000000   //R1_nF:OR:R1_iA
    mcr p15,0,r0,c1,c0,0


    /* 设置MPLLCON(0x4C000004) = (92<<12)|(1<<4)|(1<<0)

     *  m = MDIV+8 = 92+8=100
     *  p = PDIV+2 = 1+2 = 3
     *  s = SDIV = 1
     *  FCLK = 2*m*Fin/(p*2^s) = 2*100*12/(3*2^1)=400M
     */
    ldr r0, =0x4C000004
    ldr r1, =(92<<12)|(1<<4)|(1<<0)
    str r1, [r0]


    /* 一旦设置PLL, 就会锁定lock time直到PLL输出稳定

     * 然后CPU工作于新的频率FCLK

     */




    /* 设置内存: sp 栈 */

    /* 分辨是nor/nand启动

     * 写0到0地址, 再读出来

     * 如果得到0, 表示0地址上的内容被修改了, 它对应ram, 这就是

nand启动
```

```
  * 否则就是 nor 启动
  */
 mov r1, #0
 ldr r0, [r1] /* 读出原来的值备份 */
 str r1, [r1] /* 0->[0] */
 ldr r2, [r1] /* r2=[0] */
 cmp r1, r2   /* r1==r2? 如果相等表示是 NAND 启动 */
 ldr sp, =0x40000000+4096 /* 先假设是 nor 启动 */
 moveq sp, #4096  /* nand 启动 */
 streq r0, [r1]   /* 恢复原来的值 */


 bl main
halt:
 b halt
```

000 / (115200 x 16)) –1 = 26

UBRDIV0 = 26;

3. 设置数据格式

数据格式设置为常用的 8n1:8 个数据位，无较验位, 1 个停止位

ULCON0 = 0x00000003; /* 8n1: 8 个数据位，无较验位, 1 个停止位 */

读取 UTRSTAT0 寄存器，查询其第 2 位判断发送 buff 是否为空，即上一次发送是否完成，如果完成即向 UTXH0 写入要发送的新数据；查询其第 0 位判断接收 buff 是否为空，即本次接收是否完成，如果接收完成，读取 URXH0 的值。

```
int putchar(int c)
{
    /* UTRSTAT0 */
    /* UTXH0 */

    while (!(UTRSTAT0 & (1<<2)));
    UTXH0 = (unsigned char)c;

}

int getchar(void)
```

```
{
    while (!(UTRSTAT0 & (1<<0)));
    return URXH0;
}
```
循环输出字符，就可以实现字符串的输出

```
int getchar(void)
{
    while (!(UTRSTAT0 & (1<<0)));
    return URXH0;
}

int puts(const char *s)
{
    while (*s)
    {
        putchar(*s);
        s++;
    }
}
```
在主函数里，先调用初始化函数，然后循环获取用于输入的数据，然后回显出来。并且在收到`\r`回车时，输出`\n`换行，有些时候`\n`是回车，那输出`\r`换行。

```
#include "s3c2440_soc.h"
#include "uart.h"

int main(void)
{
    unsigned char c;

    uart0_init();
    puts("Hello, world!\n\r");

    while(1)
    {
        c = getchar();
        if (c == '\r')
        {
            putchar('\n');
        }

        if (c == '\n')
        {
```

```
                putchar('\r');
            }

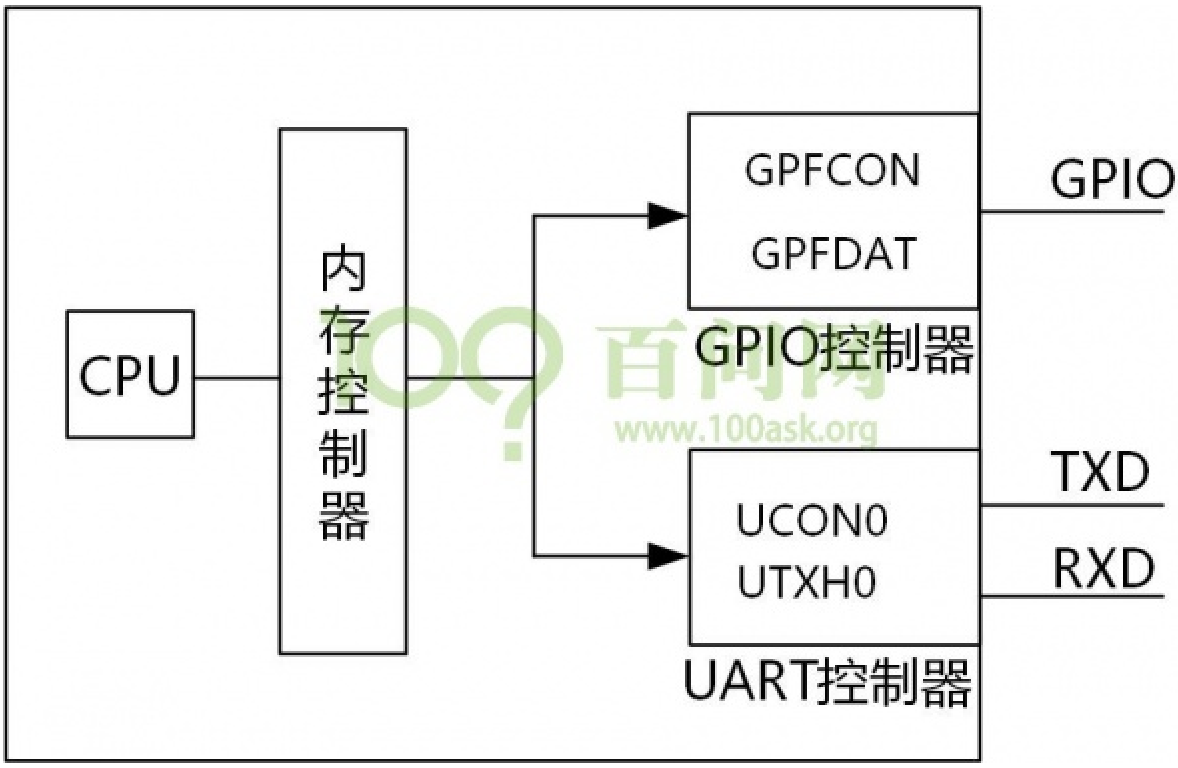
            putchar(c);
        }
        return 0;
    }
```

第 012 课 内存控制器与 SDRAM

第 001 节_辅线 1_硬件知识_内存接口概念

首先来分析下操作 GPIO 控制器和操作 UART 控制器两者的区别。

如图是 S3C2440 是个片上系统，有 GPIO 控制器（接有 GPIO 管脚）,有串口控制器 (接有 TXD RXD 引脚)。



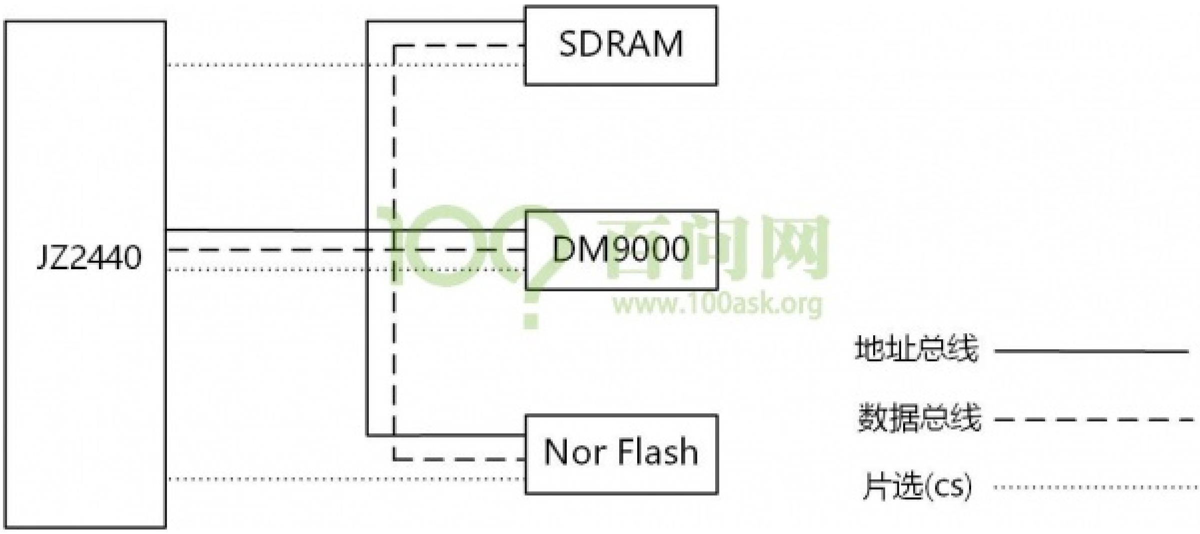
配置 GPIO 控制器相应的寄存器，即可让引脚输出高低电平；配置 UART 控制器相应的寄存器，即可让引脚输出波形。前者相对简单，类似门电路，后者相对复杂，属于协议类接口。类似的协议类接口还有 iic、iis、spi 等。 对于 CPU 是不管什么接口的，它只写相应的寄存器，由控制器根据寄存器的配置去控制具体的引脚。

那么 CPU 是如何访问各个不同的寄存器的呢？

CPU 只管发出一个地址，内存控制器根据该地址选择不同的模块，然后从模块中得到数据或者发送数据到模块中。

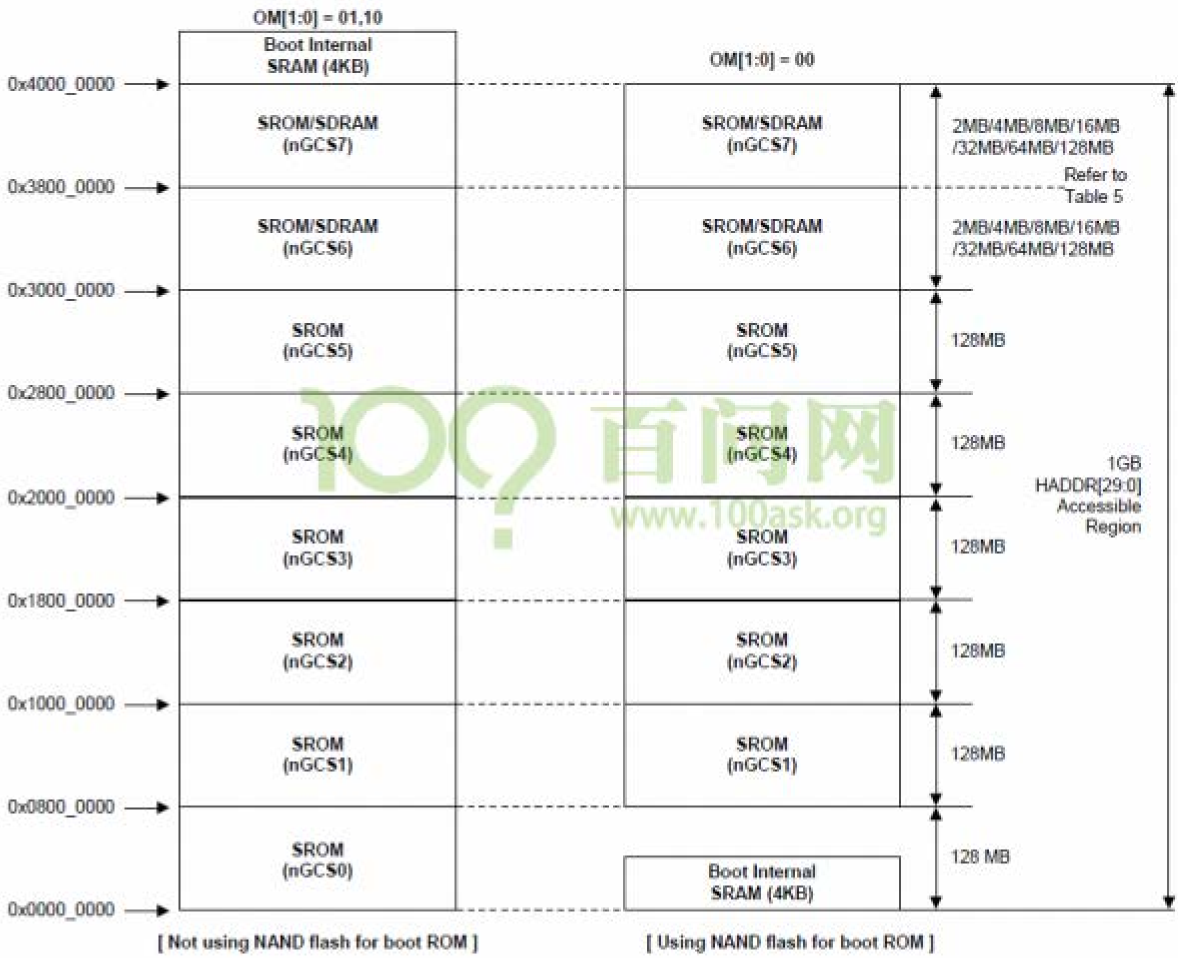
前面的 GPIO/门电路接口、协议类接口，都不会把地址输出到外部，接下来的内存类接口，会把地址输出到外部，比如 Nor Flash、网卡、SDRAM。

如图，SDRAM、DM9000 网卡、Nor Flash 都接在 JZ2440 的数据总线和地址总线上，CPU 把数据和地址发送出去，然后内存控制器根据片选信号选择相应的设备接收地址和数据信号，互不干扰。



片选信号和地址的关系怎么确定？

这个是由 2440 芯片特性决定的。



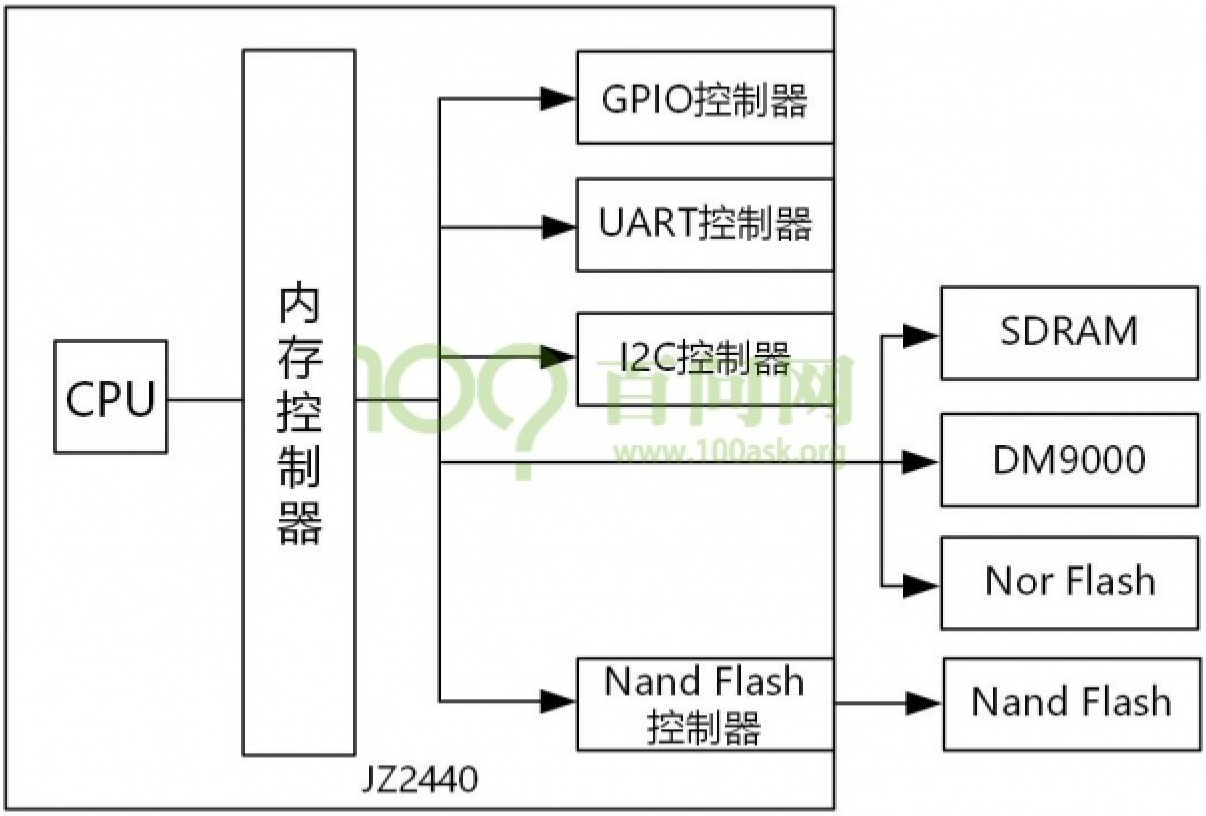
当选择 Nor Flash 启动时，CPU 发出的指令的地址范围处于 0x0000000 - 0x06000000，内存控制器就会使 nGCS0 处于低电平（片选引脚被选中），Nor Flash 被选中。

当 CPU 发出的指令的地址范围处于 0x20000000 - 0x26000000，内存控制器就会使 nGCS4 处于低电平（片选引脚被选中），网卡被选中。

当 CPU 发出的指令的地址范围处于 0x30000000 - 0x36000000，内存控制器就会使 nGCS6 处于低电平（片选引脚被选中），SDRAM 被选中。

内存控制器根据不同的地址地址范围，发出不同的片选引脚，只有被片选引脚选中的芯片才能正常工作，不被选中的芯片就像不存在一样，不工作。

GPIO/门电路接口、协议类接口、内存类接口都属于 CPU 的统一编址。对于 Nand Flash，在原理图上它的地址线并没有连接到 CPU，因此它不参与 CPU 的统一编址。但它的数据线也接到了数据总线上，为了防止干扰，它也有一个片选信号(CE)。当 CPU 访问 Nand Flash 时，Nand Flash 控制器才会片选 Nand Flash，让其接收数据总线上的数据。



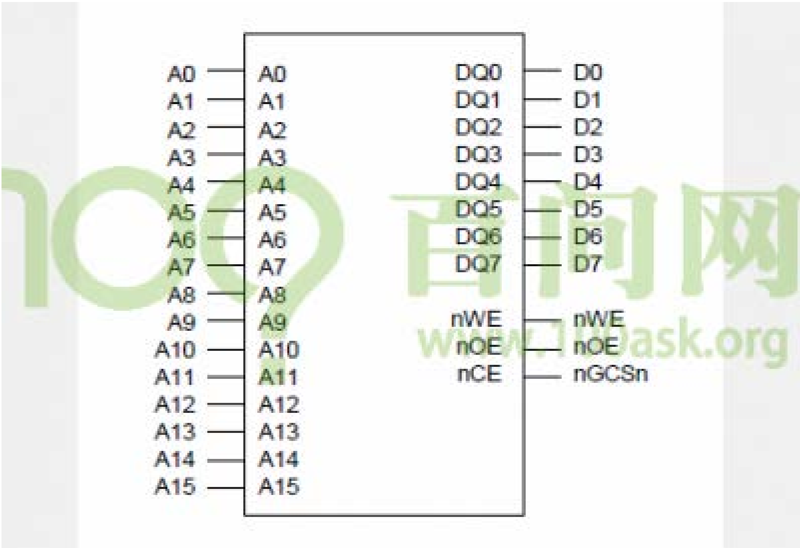
再来看下 Nor Flash 的空间，0x00000000 * 0x06000000，为 128M，即每一个片选信号可以选择的空间是 128M=2^27，也就需要 A0、A1……A26，共 27 根地址线。CPU 发出的 32 位地址线，内存控制器根据地址范围，片选上相应

的 bank，并将地址转化为 27 位。

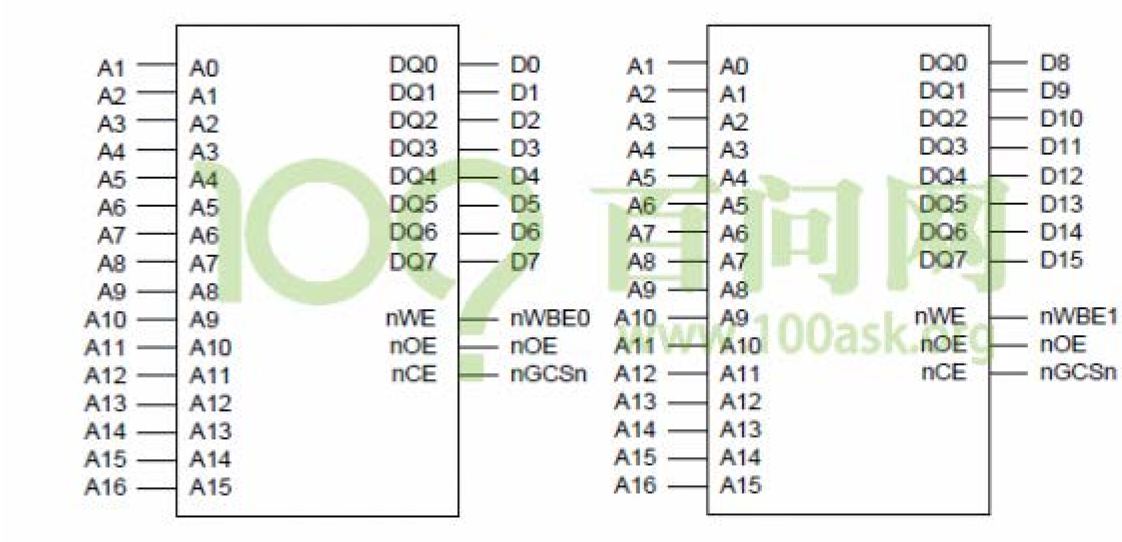


第 002 节_辅线 1_硬件知识_不同位宽设备的连接

参考 2440 芯片手册，可以看到内存接口与 8-bit ROM 连接时，2440 的 A0 与外部芯片的 A0 相连。

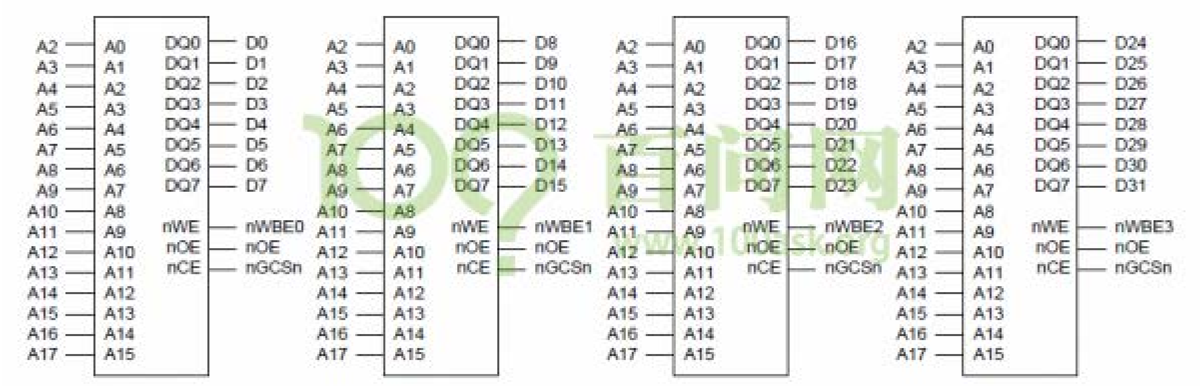


当与两个 8-bit ROM 拼接成的一个 16-bit ROM 连接时，2440 的 A1 与外部芯片的 A0 相连。

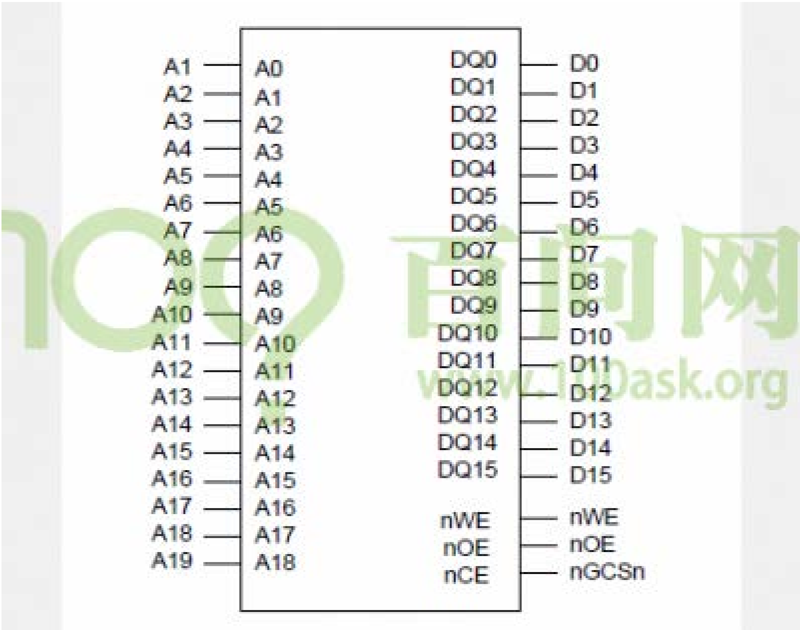


当与四个 8-bit ROM 拼接成的一个 32-bit ROM 连接时，2440 的 A2 与外部芯片

的 A0 相连。



当与一个 16-bit ROM 连接时，2440 的 A1 与外部芯片的 A0 相连。



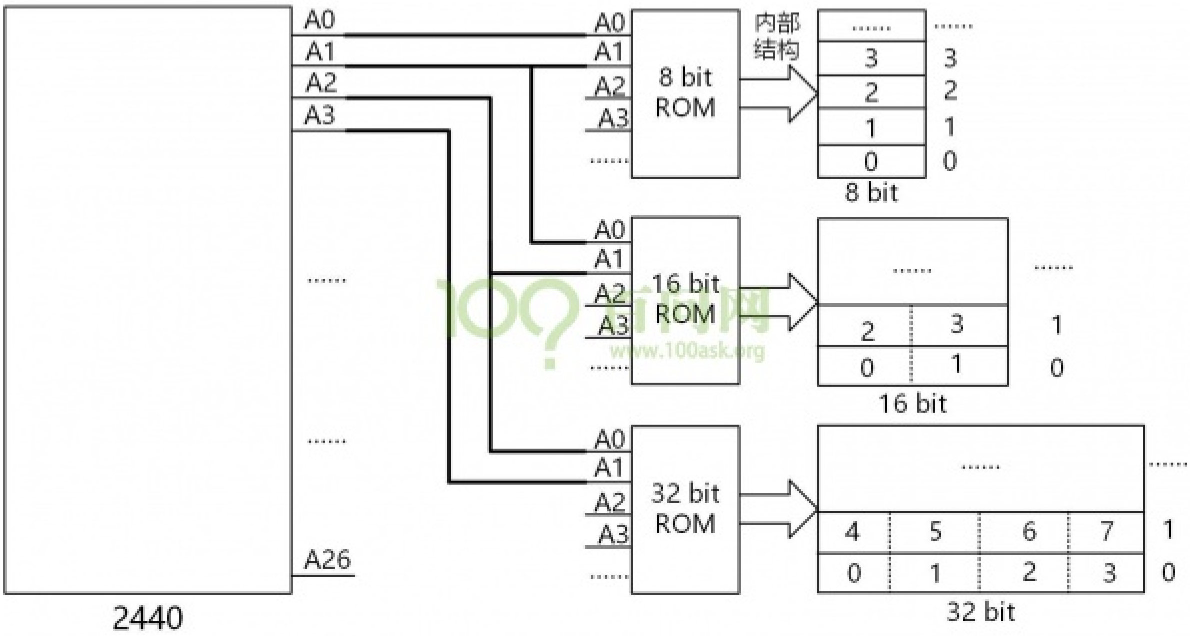
可以看出外接芯片的位宽有变化时，地址线的接法也会有变化。那这个变化有什么规律呢？

假设 CUP 执行：

MOV R0， #3 @去地址为 3 的内存上

LDRB R1, [R0] @ 从内存为 3 的地址上，读出一个字节

如图有 8bitROM、16bitROM、32bitROM。



8 个 bit 组成一个字节，字节是计算机的最小的存储单位，因此我们读取数据肯定都是 8bit 的倍数。

对于 8bitROM ，8bit 是一次读写的最小单位，即 0 地址是第一个 8bit，1 地址是第二个 8bit;CPU 发出的命令是读取地址为 3 上的数据，即 A0 和 A1 都为 1，8bitROM 的 A0 和 A1 收到的也都是 1，于是找到了 ROM 上地址为 3 的 8bit 数据,包含了我们需要的数据。

对于 16bitROM ，16bit 是一次读写的最小单位，即 0 地址是第一个 16bit，里面有两个 8bit 数据;CPU 发出的命令是读取地址为 3 上的数据，即 A0 和 A1 都为 1，16bitROM 的 A0 和 A1 分别收到的是 1 和 0，于是找到了 ROM 上地址为 1 的 16bit 数据，包含了我们需要的数据，最后内存控制器再帮我们挑选出所需的 8bit 数据。

对于 32bitROM ，32bit 是一次读写的最小单位，即 0 地址是第一个 32bit，里面有四个 8bit 数据;CPU 发出的命令是读取地址为 3 上的数据，即 A0 和 A1 都为 0，32bitROM 的 A0 和 A1 收到的都是 0，于是找到了 ROM 上地址为 0 的 32bit 数据，包含了我们需要的数据，最后内存控制器再帮我们挑选出所需的 8bit 数据。

ROM/bit	CPU 发出地址	ROM 收到地址	ROM 返回数据	内存控制器挑选出数据给 CPU
8bit（ROM)	000011	000011	编号 3 的存储单元中的 8 数据	编号 3 的存储单元中的 8bit 数据
16bit（ROM)	000011	000001	编号 1 的存储单元中的 16 数据	根据”A0=1”,挑出低 8bit 数据

32bit（ROM）	000011	000000	编号 0 的存储单元中的 32 数据	根据“A0A1=11”,挑出最低 8bit 数据

接到芯片上的引脚用来确定读取芯片上的哪一个单元的数据，把这个单元的数据返回给内存控制器，内存控制器会根据没有连接芯片的引脚，来确定返回哪一个单元的数据给 CPU,

再举一个例子： 假如传递一个 32 位的数据时

MOV R0, #4

LDR R1, [R0] @去地址 4，读取 4 字节数据

执行过程如下：

8bitROM: 当 CPU 发出地址（000100），内存控制器会把 000100,000101,000110,000111 处的地址转发给 ROM，ROM 会把得到的地址 000100,000101,000110,000111，上的数据返回给内存控制器，内存控制器会把得到的 4 个 8bit 的数据组装成一个 32 位的数据返回给 CPU。

16bitROM: 当 CPU 发出地址（000100），内存控制器会把 00010，00011 处的地址转发给 ROM，ROM 会把得到的地址 00010，00011，上的数据返回给内存控制器，内存控制器会把得到的 2 个 16bit 的数据组装成一个 32 位的数据返回给 CPU。

32bitROM: 当 CPU 发出地址(000100)，内存控制器会把 0001 处的地址发送给 ROM，ROM 会把得到的地址 0001 上的数据返回给内存控制器，内存控制器会把得到的 1 个 32bit 数据返回给 CPU。

怎样确定芯片的访问地址： 1. 根据片选信号确定基地址， 2. 根据芯片所接地址线确定范围

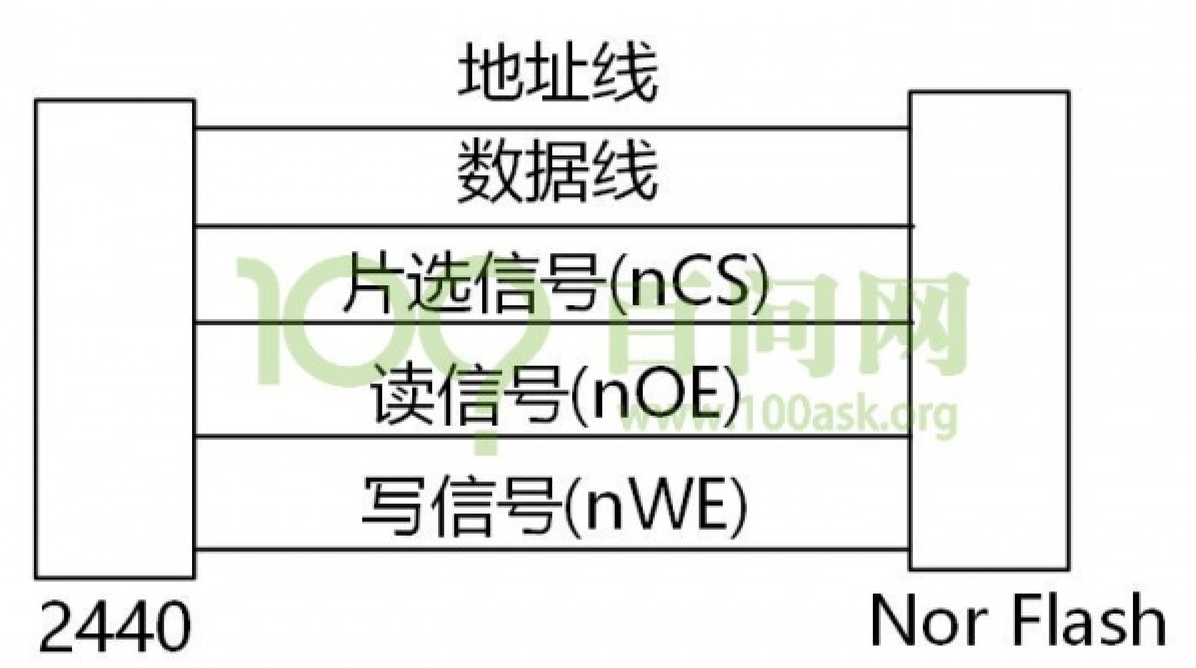
实例： Nor Flash 使用的是片选 0(nGCS0),基地址为 0，用到 A20,A19......A1，A0 共 21 条地址线，所以地址范围为 0x00000000 ~ 0x1FFFFF 也就是 2M 的空间大小。 网卡(Net)使用的是片选 4(nGCS4),基地址为 0x20000000，用到 A2,A0 共 2 根地址线，所以地址范围为 0x20000000 ~ 0x20000005。 SDRAM 使用的是片选 6(nGCS6),基地址为 0x30000000。

-	Nor Flash	Net	
基地址（base）	0x00000000	0x20000000	0x30000000
取址范围	0x00000000~0x1FFFFF	0x20000000~0x20000005	比较特殊，后面讲解

第 003 节_辅线 1_硬件知识_时序图分析示例

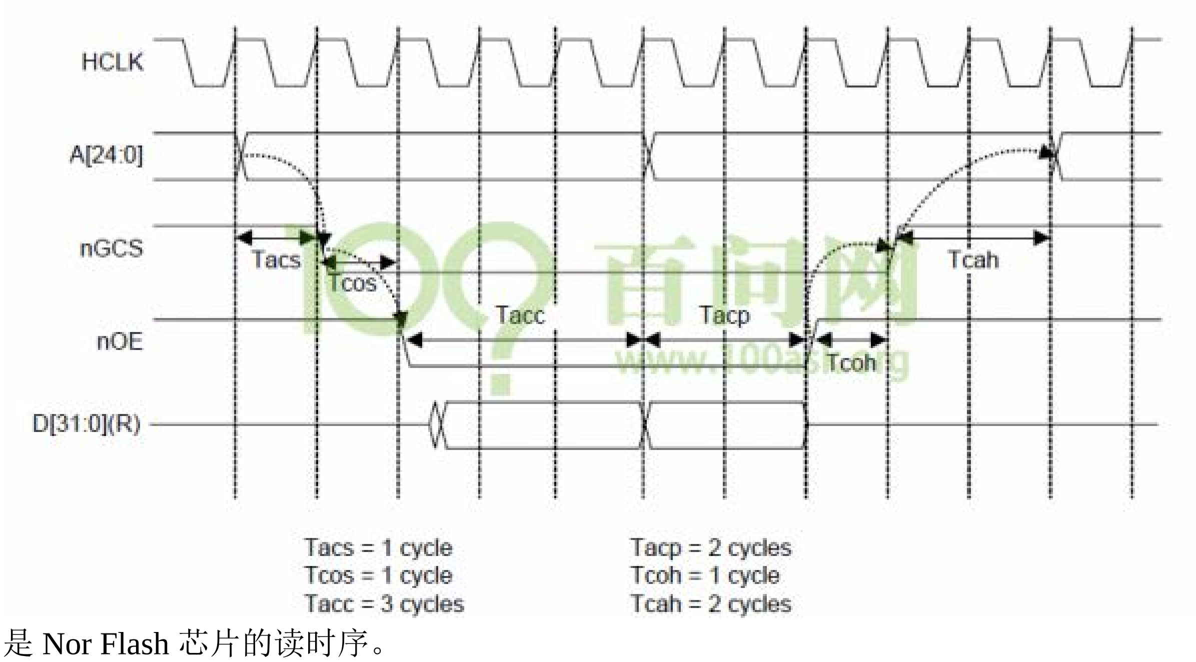
这节我们分析一下我们了解时序图，信号之间是怎样一起工作的，以 Nor Flash 为例。

2440 和 Nor Flash 之间有地址线，数据线，还有各种数据线连接。

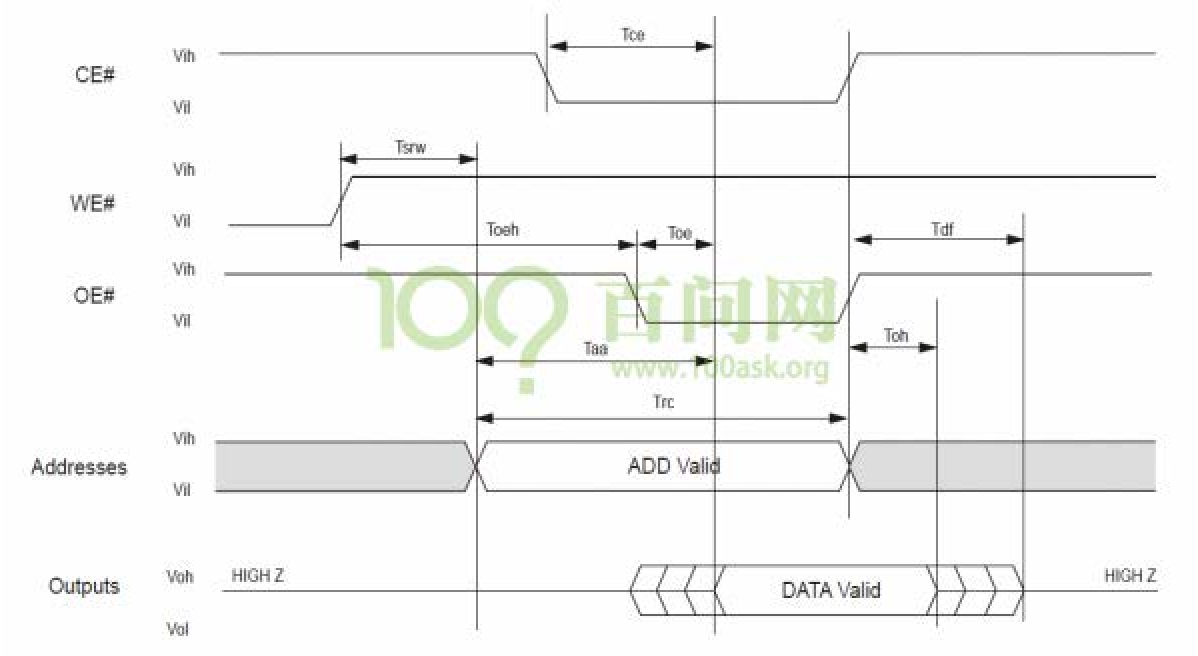


以 Nor Flash 为例，分析下如何设置它的时序。

如图是 S3C2440 的 Nor Flash 控制器的读时序图，里面很多参数都需要根据外接芯片的性能进行设置，有的芯片性能好、响应时间快，就可以把参数时间设置小一点，释放更好的性能。



如图是 Nor Flash 芯片的读时序。



我们需要做的就是设置 S3C2440 的 Nor Flash 控制器时序去满足 Nor Flash 芯片的时序。每个参数的参考范围可以通过 AC CHARACTERISTICS 得到。

AC CHARACTERISTICS

Symbol	Description	Min.	Typ.	Max.	Unit
Taa	Valid data output after address			70	ns
Tce	Valid data output after CE# low			70	ns
Toe	Valid data output after OE# low			30	ns
Tdf	Data output floating after OE# high or CE# high			30	ns
Toh	Output hold time from the earliest rising edge of address, CE#, OE#	0			ns
Trc	Read period time	70			ns
Tsrw	Latency Between Read and Write Operation (*Note 1)	45			ns
Twc	Write period time	70			ns
Tcwc	Command write period time	70			ns
Tas	Address setup time	0			ns
Tah	Address hold time	45			ns
Tds	Data setup time	35			ns

结合 Nor Flash 芯片的两张图，可以得到如下信息:

发出地址数据(Addresses)后，要等待 Taa(要求大于等于 70ns)时间，地址数据才有效;

发出片选信号(CE#)后，要等待 Tce(要求大于等于 70ns)时间，片选信号才有效;

发出读信号(OE#)后要等待 Toe(要求大于等于 30ns)时间，读信号才有效;

为了简单我们把地址数据(Addresses)，片选信号(CE#)，读信号(OE#)，同时发出，然后让它们都等待 70ns(等待信号有效)。对应 S3C2440 的 Nor Flash 控制器的读时序图，需要让地址信号 A[24:0]、片选信号 nGCS、读信号 nOE 同时发出，保持 Tacc 大于等于 70ns。

查阅 S3C2240 的参考手册，Nor Flash 是接在 BANKCON0 上的，因此只需要设置 BANKCON0 即可。

Register	Address	R/W	Description	Reset Value
BANKCON0	0x48000004	R/W	Bank 0 control register	0x0700
BANKCON1	0x48000008	R/W	Bank 1 control register	0x0700
BANKCON2	0x4800000C	R/W	Bank 2 control register	0x0700
BANKCON3	0x48000010	R/W	Bank 3 control register	0x0700
BANKCON4	0x48000014	R/W	Bank 4 control register	0x0700
BANKCON5	0x48000018	R/W	Bank 5 control register	0x0700

BANKCONn	Bit	Description	Initial State
Tacs	[14:13]	Address set-up time before nGCSn 00 = 0 clock 01 = 1 clock 10 = 2 clocks 11 = 4 clocks	00
Tcos	[12:11]	Chip selection set-up time before nOE 00 = 0 clock 01 = 1 clock 10 = 2 clocks 11 = 4 clocks	00
Tacc	[10:8]	Access cycle 000 = 1 clock 001 = 2 clocks 010 = 3 clocks 011 = 4 clocks 100 = 6 clocks 101 = 8 clocks 110 = 10 clocks 111 = 14 clocks Note: When nWAIT signal is used, Tacc ≥ 4 clocks.	111
Tcoh	[7:6]	Chip selection hold time after nOE 00 = 0 clock 01 = 1 clock 10 = 2 clocks 11 = 4 clocks	000
Tcah	[5:4]	Address hold time after nGCSn 00 = 0 clock 01 = 1 clock 10 = 2 clocks 11 = 4 clocks	00
Tacp	[3:2]	Page mode access cycle @ Page mode 00 = 2 clocks 01 = 3 clocks 10 = 4 clocks 11 = 6 clocks	00
PMC	[1:0]	Page mode configuration 00 = normal (1 data) 01 = 4 data 10 = 8 data 11 = 16 data	00

可以看到 Tacc 上电初始值是 111，对应 14 个 clocks。系统上电采用 12MHz 的晶振，HCLK=12MHz，Tacc=(1000/12*14)≈1166ns，这个值很大，几乎可以满足所有 Nor Flash 的要求。

启动后，将 HCLK 设置为 100MHz，T=1000/100=10ns，Tacc 需要大于等于 70ns，因此设置 Tacc 等于 101，8 个 clocks 即可。

在前面 uart 实验的源码基础上，新建 init.c 和 init.h 两个文件。

在 init.c 里面只需要设置 BANKCON0 寄存器即可。

```
#include "s3c2440_soc.h"

void bank0_tacc_set(int val)
{
    BANKCON0 = val << 8;
}
```

init.h 进行函数声明。

```
#ifndef _INIT_H
#define _INIT_H
```

```c
void bank0_tacc_set(int val);

#endif
```

最后在主函数里面，通过串口获取输入的值，传入 bank0_tacc_set()函数里，设置 Tacc，然后再读取 Nor Flash 上的闪灯程序。

```c
#include "s3c2440_soc.h"
#include "uart.h"
#include "init.h"

int main(void)
{
    unsigned char c;

    uart0_init();
    puts("Enter the Tacc val: \n\r");

    while(1)
    {
        c = getchar();
        putchar(c);
        if (c >= '0' && c <= '7')
        {
            bank0_tacc_set(c - '0');
            led_test();
        }
        else
        {
            puts("Error, val should between 0~7\n\r");
            puts("Enter the Tacc val: \n\r");
        }
    }
    return 0;
}
```

实验效果：

输入 0~4,Tacc 小于 70ns,无法读取 Nor Flash 上数据，LED 不能闪烁。

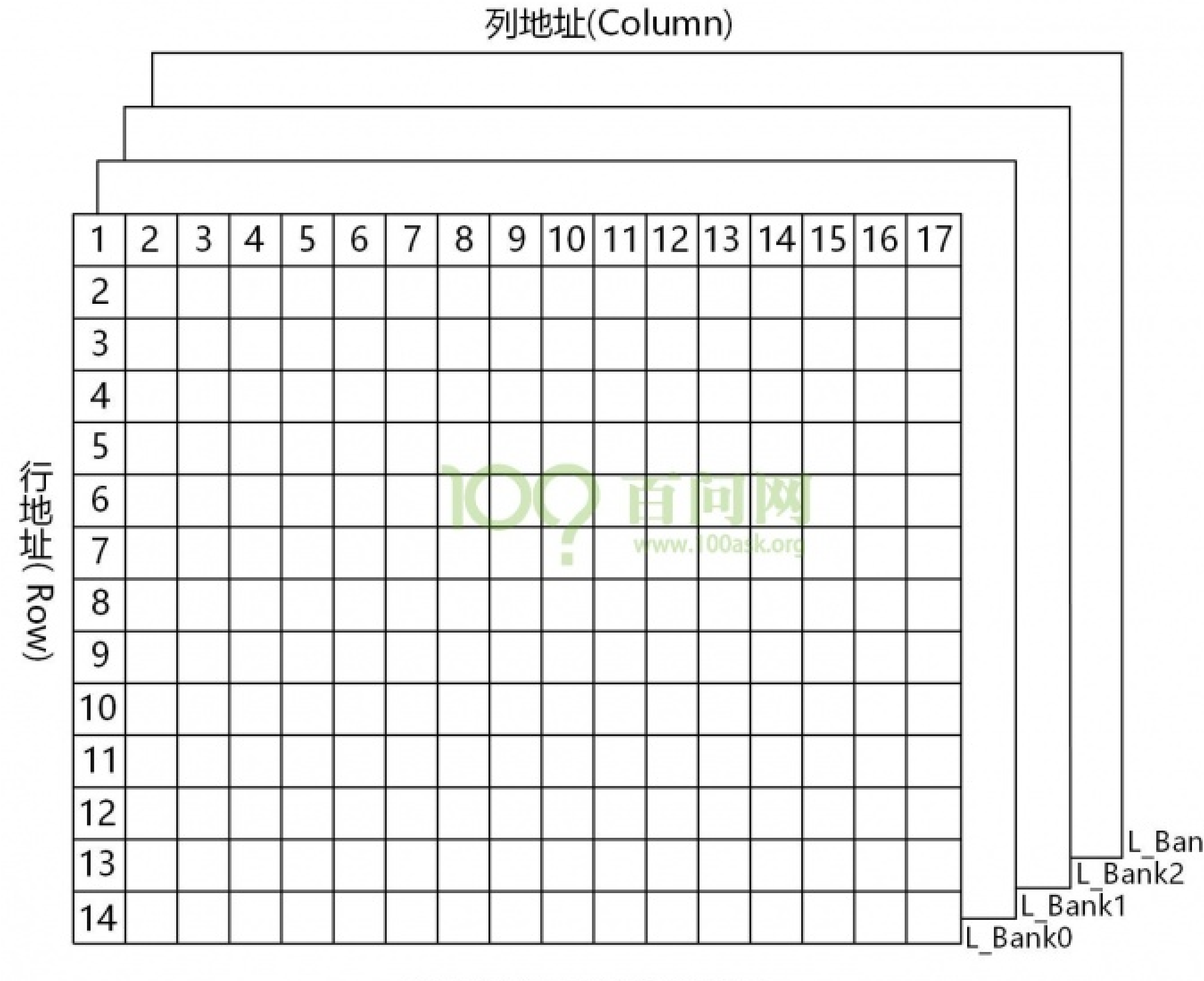
输入 4~7，Tacc 大于 70ns,可以读取 Nor Flash 上数据，LED 不断闪烁，且值越小越快(区别不明显)。

第 004 节_辅线 1_硬件知识_SDRAM 的设置

本节将讲解如何设置 SDRAM，如果想对内存有更多的了解，可以在网上搜索看下这篇文档“高手进阶_终极内存技术指南——完整/进阶版”。

在 JZ2440 上接有 64M 的 SDRAM，如果想要使用 SDRAM，需要对内存控制器做一些设置。 在前面第一节讲到，CPU 将数据或地址发给内存控制器，内存控制器再去访问外部的 SDRAM，因此设置内存控制器就说本节的核心。

如图是 SDRAM 存储结构逻辑图：



SDRAM存储结构逻辑图

SDRAM 总共有 4 个块(Banks)，可以认为每个块就是一个表格，里面的每个格子表示的是 16bit 数据。

问题 1：怎样访问里面的某个格子呢？

1. 首先发出一个片选信号，选中整个芯片;
2. 发出 Bank 地址，选择是哪一个 Bank(块，即表格);
3. 发出行地址;
4. 最后发出列地址，才能选中是个格子;

问题 2：那么多的信号有谁发出呢？

由内存控制器发出，所以我们需要设置内存控制器，CPU 只是简单的执行读写内存的命令，其他的都交给内存控制起来处理。 例如

LDR R0,=0x30000000

LDR R1,[R0]

1. CPU 把 0x30000000 这个地址发给内存控制器，内存控制器根据这个地址，判断属于哪个范围，然后发出相应的片选信号。

2. 根据类型(比如 SDRAM)拆分成三部分：发出 Bank 地址、发出行地址、发出列地址。

对于上面的三个，怎样拆分呢？

行地址线有几条，列地址线有几条，都要设置内存控制器里面相关寄存器。

3. 读数据

综上所述：对 SDRAM 的访问可以分为如下步:

1. CPU 发出的片选信号 nSCS6 有效，它选中 SDRAM 芯片。

2. SDRAM 中有 4 个 L-Bank，需要两根地址信号来选中其中之一，根据原理图，可知使用 ADDR24，ADDR25 作为 L-Bank 的选择信号。

3. 对被选中的芯片进行统一的行/列（存储单元）寻址。

根据 SDRAM 芯片的列地址线数目设置 CPU 的相关寄存器后，CPU 就会从 32 位的地址中自动分出 L_Bank 片选信号，行地址信号，列地址信号，然后先后发出行地址信号，列地址信号。L_Bank 选择信号在发出行地址信号的同时发出，并维持到列地址信号结束。

根据原理图可知：

地址、列地址共用地线 ADDR2—ADDRI4（BANK6 位宽为 32，ADDRO/I 没有使用)，使用 nSRAS、nSCAS 两个信号来区分它们。

比如本开发板中，使用两根地址线 ADDR24、ADDR25 作为 L-Bank 的选择信号：SDRAM 芯片 K4s5m632 的行地址数为 13，列地址数为 9，所以当 nSRAS 信号有时，ADDR2—ADDR14 上发出是行地址信号，它对应 32 位地址空间的 b 可 23m]：当 nSCAS 信号有效时，ADDR2—ADDR10 上发出的是列地址信号，它对应 32 位地址空间的 bit[0：2]；由于 BANK6 以 32 位的宽度外接 DRAM，ADDR0、ADDR1 恒为 0，不参与译码。

4. 找到了存储单元后，被选中的芯片就要进行统一的数据传输了。

开发板中使用两片 16 位的 SDRAM 芯片并联组成 32 位的位宽，与 CPU 的 32 根数据线(DATA0—DATA31)相连。 BANK6 的起始地址为 0x30000000，所以 SDRAM 的访问地址为 0x30000000～低 0x33FFFFFF，共 64MB。

2440 内存控制器设置：

内存控制器共有 13 个寄存器，

BANK0--BANK5 只需要设置 BWSCON 和 BANKCONx(x 为 0～5）两个寄存器；

BANK6、BANK7 外接 SDRAM 时，除 BWSCON 和 BANKCONx（x 为 6、7）外，还要设置 REFRESH、BANKSIZE、MRSRB6、MRSRB7 等 4 个寄存器。

下面分类说明各个寄存起的设置。

1. 位宽和等待控制寄存器 BWSCON(BUSWIDTH&WAITCONTROLREGISTER)

BUS WIDTH & WAIT CONTROL REGISTER (BWSCON)

Register	Address	R/W	Description	Reset Value
BWSCON	0x48000000	R/W	Bus width & wait status control register	0x000000

BWSCON	Bit	Description	Initial state
ST7	[31]	Determines SRAM for using UB/LB for bank 7. 0 = Not using UB/LB (The pins are dedicated nWBE[3:0]) 1 = Using UB/LB (The pins are dedicated nBE[3:0])	0
WS7	[30]	Determines WAIT status for bank 7. 0 = WAIT disable 1 = WAIT enable	0
DW7	[29:28]	Determines data bus width for bank 7. 00 = 8-bit 01 = 16-bit, 10 = 32-bit 11 = reserved	0
ST6	[27]	Determines SRAM for using UB/LB for bank 6. 0 = Not using UB/LB (The pins are dedicated nWBE[3:0) 1 = Using UB/LB (The pins are dedicated nBE[3:0])	0
WS6	[26]	Determines WAIT status for bank 6. 0 = WAIT disable, 1 = WAIT enable	0
DW6	[25:24]	Determines data bus width for bank 6. 00 = 8-bit 01 = 16-bit, 10 = 32-bit 11 = reserved	0

BWSCON 中每 4 位控制一个 BANK，最高 4 位对应 BANK7(没有使用)、接下来 4 位对应 BANK6。

（1）ST6[27]: 启动/禁止 SDRAM 的数据掩码引脚，对于 SDRAM，此位为 0：对于 SRAM 此位为 1。

（2）WS6[26]：是否使用存储器的 WAIT 信号，通常设为 0。

（3）DW6[25:24]：使用两位来设置相应 BANK 的位宽，0b00 对应 8 位，0b01 对应 16 位，0b10 对应 32 位(开发板用的就是 32 位的)，0b11 表示保留。

因此 BWSCON 寄存器的值为：0x22000000。

2. BANK 控制寄存器 BANKCON6（BANKCONTROLREGISTER）

在 8 个 BANK 中，只有 BANK6 和 BANK7 可以外接 SRAM 或 SDRAM。

BANK CONTROL REGISTER (BANKCONn: nGCS6-nGCS7)

Register	Address	R/W	Description	Reset Value
BANKCON6	0x4800001C	R/W	Bank 6 control register	0x18008
BANKCON7	0x48000020	R/W	Bank 7 control register	0x18008

BANKCONn	Bit	Description	Initial State
MT	[16:15]	Determine the memory type for bank6 and bank7. 00 = ROM or SRAM 01 = Reserved (Do not use) 10 = Reserved (Do not use) 11 = Sync. DRAM	11

(1)MT[16:15]：用于设置本 BANK 外接的是 ROM/SRAM 还是 SDRAM,SRAM：0b00，SDRAM：0b11(开发板使用的是 SDRAM)。

当 MT[16:15]设置为 00 时，此寄存器与 BANKC0N0、BANKCON5 类似，不再赘述。

当 MT[16:15]设置为 11 时，此寄存器其他值如下设置。

Memory Type = SDRAM [MT=11] (4-bit)			
Trcd	[3:2]	RAS to CAS delay 00 = 2 clocks 01 = 3 clocks 10 = 4 clocks	10
SCAN	[1:0]	Column address number 00 = 8-bit 01 = 9-bit 10= 10-bit	00

（2）Trcd[3:2]：行地址和列地址间隔多长时间，看芯片手册时间间隔是 20ns，本来开发板的 HCLK 是 100MHZ,clocks 为 10ns,所以设置为推荐值 0b00(2clocks)。

（3）SCAN[1:0]：SDRAM 的列地址位数，对于本开发板使用的 SDRAMK4S561632。列地址位数为 9，所以 SCAN=0b010 如果使用其他型号的 SDRAM，需要查看其数据手册。来决定 SCAN 的取值。0b00 表示 8 位，0b01 表示 9 位，0b10 表示 10 位。

综上所述，本开发板中 BANKCON6 设为 0x017001。

3. 刷新控制寄存器 REFRESH(REFRESHCONTROLREGISTER)

REFRESH CONTROL REGISTER

Register	Address	R/W	Description	Reset Value
REFRESH	0x48000024	R/W	SDRAM refresh control register	0xac0000

REFRESH	Bit	Description	Initial State
REFEN	[23]	SDRAM Refresh Enable 0 = Disable 1 = Enable (self or CBR/auto refresh)	1
TREFMD	[22]	SDRAM Refresh Mode 0 = CBR/Auto Refresh 1 = Self Refresh In self-refresh time, the SDRAM control signals are driven to the appropriate level.	0
Trp	[21:20]	SDRAM RAS pre-charge Time 00 = 2 clocks 01 = 3 clocks 10 = 4 clocks 11 = Not support	10
Tsrc	[19:18]	SDRAM Semi Row cycle time 00 = 4 clocks 01 = 5 clocks 10 = 6 clocks 11 = 7 clocks SDRAM Row cycle time: Trc=Tsrc+Trp If Trp = 3clocks & Tsrc = 7clocks, Trc = 3+7=10clocks.	11
Reserved	[17:16]	Not used	00
Reserved	[15:11]	Not used	0000
Refresh Counter	[10:0]	SDRAM refresh count value. Refer to chapter 6 SDRAM refresh controller bus priority section. Refresh period = (2^{11}-refresh_count+1)/HCLK Ex) If refresh period is 7.8 us and HCLK is 100MHz, the refresh count is as follows: Refresh count = 2^{11} + 1 - 100x7.8 = 1269	0

(1)REFEN[23]：0=禁止 SDRAM 的刷新功能，1：开启 SDRAM 的刷新功能(设置开启 SDRAM 的刷新功能)。

（2）TREFMD[22]：SDRAM 的刷新模式，0=CBR/AutoRefresh，1=SelfRefresh（一般在系统休眠时使用），我们设置默认值。

（3）Trp[21：20)：根据芯片手册设为 0 即可。

（4）Tsrc[19：18]:根据芯片手册设为默认值 0b01 即可。

（5）RefreshCounter[10:0]：即 R_CNT

R_CNT 可如下汁算（SDRAM 时钟频率就是 HCLK):

R_CNT=2^11+1-SDRAM 时钟频率(MZ)*SDRAM 刷新周期(us)

SDRAM 的刷新周期在 SDRAM 的数据手册上有标明，在本开发板使用的 SDRAM:K4S561632 的数据手册上，可看见这么一行“64msrefreshpenod(8KCycle)所以，刷新周期=64ms/8192=7.8125us。

Refreshcount=2^11+1-100x7.8=1269=0x4F5。

因此，本开发板中 REFRESH 设为 0x8404F5。

4. BANKSIZE 寄存器 REFRESH（BANKSIZEREG ISTER）

BANKSIZE REGISTER

Register	Address	R/W	Description	Reset Value
BANKSIZE	0x48000028	R/W	Flexible bank size register	0x0

BANKSIZE	Bit	Description	Initial State
BURST_EN	[7]	ARM core burst operation enable. 0 = Disable burst operation. 1 = Enable burst operation.	0
Reserved	[6]	Not used	0
SCKE_EN	[5]	SDRAM power down mode enable control by SCKE 0 = SDRAM power down mode disable 1 = SDRAM power down mode enable	0
SCLK_EN	[4]	SCLK is enabled only during SDRAM access cycle for reducing power consumption. When SDRAM is not accessed, SCLK becomes 'L' level. 0 = SCLK is always active. 1 = SCLK is active only during the access (recommended).	0
Reserved	[3]	Not used	0
BK76MAP	[2:0]	BANK6/7 memory map 010 = 128MB/128MB 001 = 64MB/64MB 000 = 32M/32M 111 = 16M/16M 110 = 8M/8M 101 = 4M/4M 100 = 2M/2M	010

(1)BURST_EN[7]：0=ARM 核禁上突发传输，1=ARM 核支持突发传输(推荐);

(2)SCKEEN[5]：0=不使用 SCKE 信号令 SDRAM 进入省电模式，1=使用 SCKE 信号令 SDRAM 进入省电模式(推荐);

(3)SCLK-EN[4]：0=时刻发出 SCLK 信号，1=仅在访问 SDRAM 期间发出 SCLK 信号（推荐);

(4)BK76MAP[2:0]：设置 BANK6 的大小。本开发板 BANK6 外接 64MB 的 SDRAM，令[2:0]=b001（64M/64M），表示 BANK6/7 的容量都是 64MB，虽然 BANK7 没有使用。

因此，本开发板中 BANKSIZE 设为 0xB1。

5. SDRAM 模式设置寄存器 MRSRBx6(SDRAM MODE REGISTER SET REGISTER）

SDRAM MODE REGISTER SET REGISTER (MRSR)

Register	Address	R/W	Description	Reset Value
MRSRB6	0x4800002C	R/W	Mode register set register bank6	xxx
MRSRB7	0x48000030	R/W	Mode register set register bank7	xxx

MRSR	Bit	Description	Initial State
Reserved	[11:10]	Not used	–
WBL	[9]	Write burst length 0: Burst (Fixed) 1: Reserved	x
TM	[8:7]	Test mode 00: Mode register set (Fixed) 01, 10 and 11: Reserved	xx
CL	[6:4]	CAS latency 000 = 1 clock, 010 = 2 clocks, 011=3 clocks Others: reserved	xxx
BT	[3]	Burst type 0: Sequential (Fixed) 1: Reserved	x
BL	[2:0]	Burst length 000: 1 (Fixed) Others: Reserved	xxx

NOTE: MRSR register must not be reconfigured while the code is running on SDRAM.
IMPORTANT NOTE: In sleep mode, sdram has to enter sdram self-refresh mode.

能修改的只有位 CL[6:4],这是 SDRAM 时序的一个时间参数，表示发出行、列地址后，等多久才返回收到数据， CL 可以取值为 0b0l0（2 clocks）或 0b011（3 clocks）。

本开发板取最保守的值 0b010,所以 MRSRB6 的值为 0x20。

下面开始写程序：

在 init.c 里面进行对内存控制器的寄存器依次进行设置：

```
void sdram_init(void)
{
    BWSCON = 0x22000000;

    BANKCON6 = 0x17001;
    BANKCON7 = 0x17001;

    REFRESH   = 0x8404f5;

    BANKSIZE = 0xb1;

    MRSRB6    = 0x20;
    MRSRB7    = 0x20;
}
```

再写一个测试函数，向 SDRAM 里面连续写 1000 个数，再读出数据对比是否是设置的数，返回对比结果：

```
int sdram_test(void)
{
```

```
    volatile unsigned char *p = (volatile unsigned char *)0x30000000;
    int i;

    // write sdram
    for (i = 0; i < 1000; i++)
        p[i] = 0x55;

    // read sdram
    for (i = 0; i < 1000; i++)
        if (p[i] != 0x55)
            return -1;

    return 0;
}
```

在主函数里调用 sdram_test()测试函数，如果测试成功，LED 闪烁：

```
int main(void)
{
    uart0_init();

    sdram_init();

    if (sdram_test() == 0)
        led_test();

    return 0;
}
```

实验结果：

LED 按预期闪烁，屏蔽掉 sdram_init()后，LED 不闪烁。

第 013 课 代码重定位

第 001 节_段的概念_重定位的引入

S3C2440 的 CPU 可以直接给 SDRAM 发送命令、给 Nor Flash 发送命令、给 4K 的片上 SDRAM 发送命令，但是不能直接给 Nand Flsh 发送命令

假如把程序烧写到 Nand Flsh 上，即向 Nand Flsh 烧入 bin 文件，CPU 是无法从 Nand Flsh 中取代码执行的。

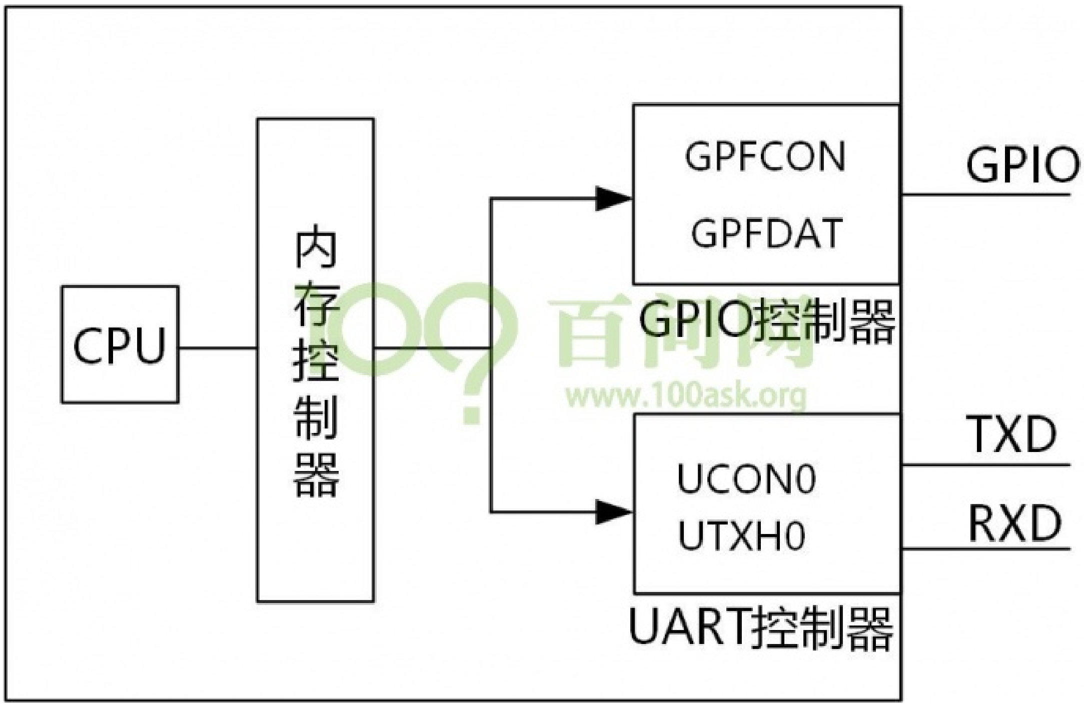
为什还可以使用 NAND 启动？

1. 上电后，Nand 启动硬件会自动把 Nand Flsh 前 4K 复制到 SRAM；
2. CPU 从 0 地址运行 SRAM；

如果我的程序大于 4K 怎么办？

前 4K 的代码需要把整个程序读出来放到 SDRAM(即代码重定位)。

如果从 Nor Flash 启动，会出现什么问题？
将拨动开关拨到 Nor Flash 启动时，此时 CPU 认为的 **0 地址** 在 Nor Flash 上面，片内内存 SRAM 的基地址就变成了 0x40000000(Nand 启动时片内内存 SRAM 的基地址基地址是 0)，由于 Nor Flash 特性：**可以像内存一样读，但不能像内存直接写**，因此需要把全局变量和静态变量重定位 放到 SDRAM 里。



例如执行如下几条汇编指令

```
MOV R0, #0
LDR R1, [R0] @读有效
STR R1, [R0] @写无效
```

当程序中含有需要写的全局变量或静态变量时，假如是在 Nand Flash 可以正常操作，如果是在 Nor Flash，修改无效。因此我们需要把全局变量和静态变量重定位 放到 SDRAM

```
#include "s3c2440_soc.h"
#include "uart.h"
#include "init.h"

char g_Char = 'A';     //定义一个全局变量

const char g_Char2 = 'B'; //定义固定的全局变量

int g_A = 0;
int g_B;

int main(void)
```

```
{
    uart0_init();

    while (1)
    {
            putchar(g_Char);  /*让g_Char输出*/

            g_Char++;         /* nor 启动时, 此代码无效 */
            delay(1000000);
    }


    return 0;
}
```

编译运行查看是否有效果

查看 sdram.dis 文件 发现 data 数据段放在了 0x00008474 这个地址导致程序太大

在 makefile 中加入这么一句话

arm-linux-ld -Ttext 0 **-Tdata 0x700** start.o led.o uart.o init.o main.o -o sdram.elf

16 进制的 700 就是十进制的 2048 这时我们的 bin 文件就变为 2049

烧写程序:

烧写在 NORFlash 和 烧写在 NANDFlash 观察这两种的效果。

设置成 NANDFlash 启动没有问题 显示 ABCDE...

设置成 NORFlash 启动显示 AAA...

对于 NOR 启动时 `g_Char++; /* nor 启动时, 此代码无效 */`

```
Disassembly of section .data:

00000700 <__data_start>:
 700:      Address 0x700 is out of bounds.   //数据段

Disassembly of section .rodata:
                                             //放在只读数据段内
```

```
00000474 <g_Char2>:           //const char g_Char2 = 'B';
 474:     Address 0x474 is out of bounds.

Disassembly of section .bss: //bss 段

00000804 <g_A>:                        //int g_A = 0;

 804:     00000000     andeq  r0, r0, r0

00000808 <g_B>:                        //int g_B;
 808:     00000000     andeq  r0, r0, r0
Disassembly of section .comment:
```

一个程序里面有

- .text 代码段
- .data 数据段
- rodata 只读数据段(const 全局变量)
- bss 段 (初始值为 0，无初始值的全局变量)
- commen 注释

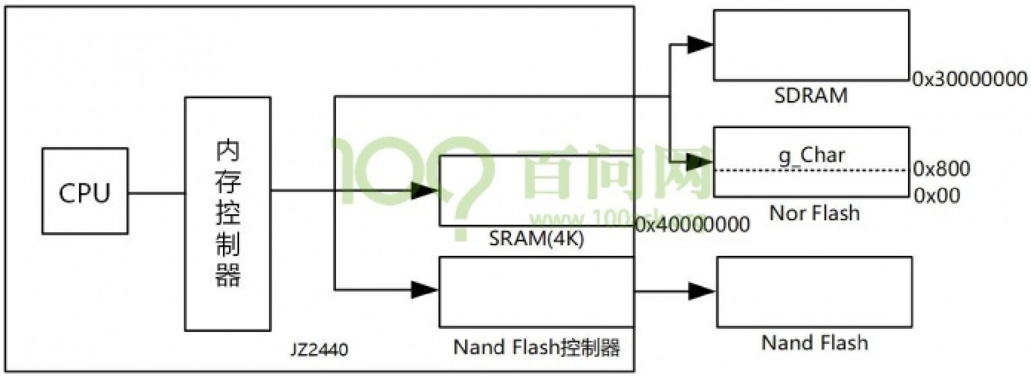
其中 bss 段和 commen 注释不保存在 bin 文件中。

第 002 节_链接脚本的引入与简单测试

前面程序运行，发现从 Nand Flash 启动和从 Nor Flash 启动表现是不一样的。
设置成 Nand Flash 启动没有问题 显示 ABCDE...
设置成 NOor Flash 启动则显示 AAA...

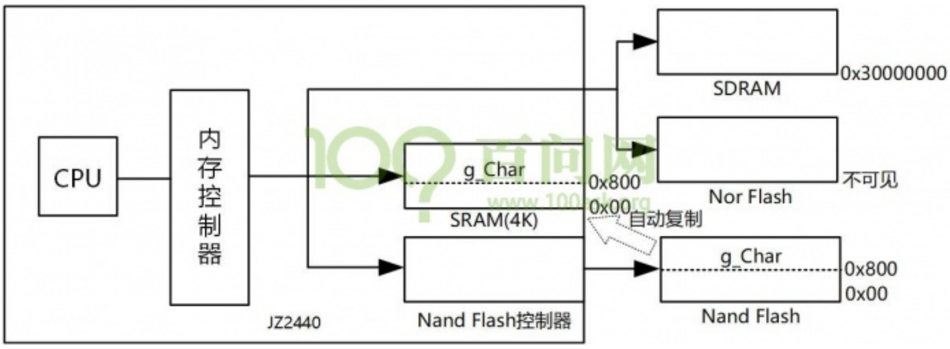
这是什么原因呢？

- 假如现在是 Nor 启动：



Nor Flash 就被认为是 0 地址，g_Char 被放在 0x700 后面。CPU 上电后从 0 地址开始执行，它能读取 Nor Flash 上的代码，打印出 A，当进行 g_Char++的时候，写操作操作无效，下次读取的数据仍然是 A。

- 假如现在是 Nor 启动：



上电后，Nand Flash 前 4K 代码就被自动的复制到 SRAM 里面，SRAM 是 CPU 认为的 0 地址。CPU 上电后从 0 地址开始执行，它读取 SRAM 上的代码，并 g_Char++修改变量，下次读取的数据就依次增加了。

为了解决 Nor Flash 里面的变量不能写的问题，我们把变量所在的数据段放在 SDRAM 里面，看行不行。 修改 Makefile 指定数据段为 0x30000000 -Tdata 0x30000000:

```
arm-linux-ld -Ttext 0 -Tdata 0x30000000  start.o led.o uart.o init.o main.o -o sdram.elf
```

这样的话编译出来的 bin 文件 从 0 地址 到 0x30000000 地址 文件大小有 700 多 MB 代码段和数据段直接有间隔 称之为黑洞



解决黑洞有两个办法：

- 第一个方法

1. 把数据段的 g_Char 和代码段靠在一起；
2. 烧写在 Nor Flash 上面；
3. 运行时把 g_char(全局变量)复制到 SDRAM，即 0x3000000 位置(重定位)；

- 第二个方法

1. 让文件直接从 0x30000000 开始，全局变量在 0x3......；
2. 烧写 Nor Flash 上 0 地址处；
3. 运行会把整个代码段数据段(整个程序)从 0 地址复制到 SDRAM 的 0x30000000(重定位)；

这两个方法的区别是前者只重定位了数据段，后者重定位了数据段和代码段。

参考文档 Using LD, the GNU linker

第一种办法如何实现 修改 Makefile 的代码段地址，使用链接脚本 sdram.lds 指定。

#arm-linux-ld -Ttext 0 -Tdata 0x30000000 start.o led.o uart.o init.o main.o -o sdram.elf

arm-linux-ld -T sdram.lds start.o led.o uart.o init.o main.o -o sdram.elf

链接脚本的语法：

```
SECTIONS {
...
secname start BLOCK(align) (NOLOAD) : AT ( ldadr )
  { contents } >region :phdr =fill
...
}
```

我们需要依次排列 代码段、只读数据段、数据段、.bss 段、.common。其中数据段放在 0x700,但运行时在 0x3000000：

```
SECTIONS {
   .text   0  : { *(.text) }//所有文件的.text

   .rodata  : { *(.rodata) } //只读数据段

   .data 0x30000000 : AT(0x700) { *(.data) } //放在0x700,但运行时在0x3000000

   .bss  : { *(.bss) *(.COMMON) }//所有文件的bss段，所有文件的.COMMON段
}
```

重新编译后烧写 bin 文件，发现启动后显示乱码。原因是我们从 0x30000000 处获取 g_Char，但在这之前，并没有在 0x30000000 处准备好数据。因此需要重定位数据段，将 0x700 的数据移动到 0x30000000 处，在 start.S 加入：

```
bl sdram_init

/* 重定位 data 段 */
mov r1, #0x700
ldr r0, [r1]
mov r1, #0x30000000
str r0, [r1]
```

```
bl main
```

上面的这种方法，只能复制 0x700 处的一位数据，不太通用，下面写一个更加通用的复制方法：

链接脚本修改如下：

```
SECTIONS {
   .text   0  : { *(.text) }
   .rodata  : { *(.rodata) }
   .data 0x30000000 : AT(0x700)
   {
      data_load_addr = LOADADDR(.data);

      data_start = . ;//等于当前位置

      *(.data)  //等于数据段的大小

      data_end = . ;//等于当前位置

   }
   .bss  : { *(.bss) *(.COMMON) }
}
```

修改 start.S

```
   bl sdram_init

   /* 重定位 data 段 */

   ldr r1, =data_load_addr  /* data 段在 bin 文件中的地址, 加载地址 */

   ldr r2, =data_start      /* data 段在重定位地址, 运行时的地址 */

   ldr r3, =data_end        /* data 段结束地址 */

cpy:

   ldrb r4, [r1] //从 r1 读到 r4

   strb r4, [r2] //r4 存放到 r2
```

```
add r1, r1, #1 //r1+1
add r2, r2, #1 //r2+1

cmp r2, r3 //r2 r3 比较

bne cpy //如果不等则继续拷贝

bl main
```

第 003 节_链接脚本的解析

链接脚本的语法

```
SECTIONS {
...
secname start BLOCK(align) (NOLOAD) : AT ( ldadr )
  { contents } >region :phdr =fill
...
}
```

解释:

secname ：段名
start ：起始地址：运行时的地址(runtime addr)；重定位地址(relocate addr)
AT (ldadr) ：可有可无(load addr:加载地址) 不写时 LoadAddr = runtime addr
{ contents } 的内容:
start.o //内容为 start.o 文件
*（.text）所有的代码段文件
start.o *(.text)文件

elf 文件格式
1 链接得到 elf 文件，含有地址信息(load addr)
2 使用加载器

2.1 对于裸板是 JTAG 调试工具
2.2 对于 APP，加载器也是 APP 把 elf 文件解析读入内存的加载地址

3 运行程序
4 如果 loadaddr != runtimeaddr 程序本身要重定位
核心程序运行时应该位于 runtimeaddr（reloate addr）或者链接地址

bin 文件

1 elf 生成 bin 文件
2 硬件机制启动
3 如果 bin 文件所在位置 不等于 runtimeaddr ，程序本身实现重定位

bin 文件/elf 文件都不保存 bss 段 这些都是初始值为 0 或者没有初始化的全局变量
程序运行时把 bss 段对应的空间清零
做个实验，把全局变量 g_A 以 16 进制打印出来

```
/* 0xABCDEF12 */
void printHex(unsigned int val)
{
   int i;
   unsigned char arr[8];

   /* 先取出每一位的值 */
   for (i = 0; i < 8; i++)
   {
          arr[i] = val & 0xf;
          val >>= 4;   /* arr[0] = 2, arr[1] = 1,
arr[2] = 0xF */
   }

   /* 打印 */
   puts("0x");
   for (i = 7; i >=0; i--)
   {
          if (arr[i] >= 0 && arr[i] <= 9)
                 putchar(arr[i] + '0');
          else if(arr[i] >= 0xA && arr[i] <= 0xF)
                 putchar(arr[i] - 0xA + 'A');
   }
}
//打印初始值为 0 的变量
int g_A = 0;
int g_B;

int main(void)
{
   uart0_init();
```

```
puts("\n\rg_A = ");
printHex(g_A);
puts("\n\r");
```

上述代码，没有清理 bss 段 g_A 等于莫名奇妙的值 并不等于 0 所以需要清理 bss 段

修改 lds 链接文件

```
SECTIONS {
   .text   0  : { *(.text) }
   .rodata  : { *(.rodata) }
   .data 0x30000000 : AT(0x700)
   {
      data_load_addr = LOADADDR(.data);
      data_start = . ;
      *(.data)
      data_end = . ;
   }

   bss_start = .; //bss 开始地址是当前位置
   .bss  : { *(.bss) *(.COMMON) }
   bss_end = .; //bss 结束地址也是当前位置
}
```

修改 start.s，清除 bss 段

```
    /* 清除 BSS 段 */
    ldr r1, =bss_start
    ldr r2, =bss_end
    mov r3, #0
clean:
    strb r3, [r1]
    add r1, r1, #1
    cmp r1, r2
    bne clean

    bl main

halt:
    b halt
```

现在的代码全局变量就是为 0，通过几行代码，就可以少几十个甚至上千个全局变量的存储空间。

第 004 节_拷贝代码和链接脚本的改进

本节进行拷贝代码的改进和链接脚本的改进。

前面重定位时，需要 ldrb 命令从的 Nor Flash 读取 1 字节数据，再用 strb 命令将 1 字节数据写到 SDRAM 里面。

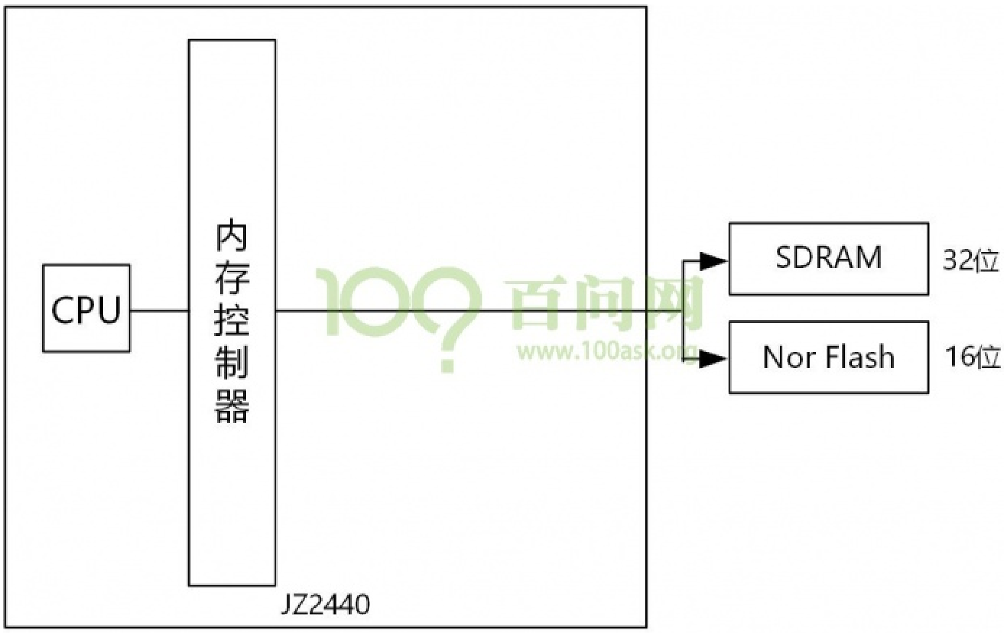
```
cpy:
    ldrb r4, [r1] /*首先从 flash 读出一个字节*/

    strb r4, [r2] /*让后把数据写到 SDRAM*/
    add r1, r1, #1
    add r2, r2, #1
    cmp r2, r3
    bne cpy
```

JZ2440 上的 Nor Flash 是 16 位，SDRAM 是 32 位。 假设现在需要复制 16byte 数据， 采用 ldrb 命令每次只能加载 1byte，因此 CPU 需要发出 16 次命令，内存控制器每次收到命令后，访问硬件 Nor Flash，因此需要访问硬件 16 次；同理，访问 SDRAM 时，CPU 需要执行 strb 16 次，内存控制器每次收到命令后，访问硬件 SDRAM，也要 16 次，这样总共访问 32 次。

现在对其进行改进，使用 ldr 从 Nor Flash 中读，ldr 命令每次加载 4 字节数据，因此 CPU 只需执行 4 次，但由于 Nor Flash 是 16 位的，内存控制器每次收到 CPU 命令后，需要拆分成两次访问，因此需要访问硬件 8 次； 使用 str 写 SDRAM，CPU 只需执行 4 次，内存控制器每次收到命令后，直接硬件访问 32 位的 SDRAM，因此这里只需要 4 次，这样总共访问只需要 12 次。 在整个操作中，花费时间最长的就是硬件访问，改进后代码，减少了硬件访问的次数，

极大的提高了效率。



根据上面原理修改代码，修改 start.S：

```
cpy:
    ldr r4, [r1]
    str r4, [r2]

    add r1, r1, #4 //r1 加 4

    add r2, r2, #4 //r2 加 4

    cmp r2, r3 //如果 r2 =< r3 继续拷贝

    ble cpy


/* 清除 BSS 段 */

    ldr r1, =bss_start
    ldr r2, =bss_end
    mov r3, #0
clean:
    str r3, [r1]
    add r1, r1, #4

    cmp r1, r2 //如果 r1 =< r2 则继续拷贝

    ble clean

    bl main
```

然后编译烧写，发现启动后没有输出字符。修改主程序，尝试以整数格式输出字符，发现输出的数从 0 开始，应该是 全局变量被破坏了。

屏蔽掉 start.S 里面的清理命令，测试是否是清除 bss 段是清除了全局变量。

```
clean:
    //str r3, [r1] //注释掉此句话，str 不仅把 bss 段清除，把全局变量这些也清除了
    add r1, r1, #4
    cmp r1, r2
    ble clean

    bl main
```

屏蔽后，正常输出，锁定了问题大致位置。查看反汇编文件，原来是没有向 4 取整。 修改链接脚本让 bss 段，使用 ALIGN(4)向 4 取整。

```
SECTIONS {
   .text   0  : { *(.text) }
   .rodata  : { *(.rodata) }
   .data 0x30000000 : AT(0x700)
   {
      data_load_addr = LOADADDR(.data);
      . = ALIGN(4);
      data_start = . ;
      *(.data)
      data_end = . ;
   }

   . = ALIGN(4);//让当前地址向 4 对齐

   bss_start = .;
   .bss  : { *(.bss) *(.COMMON) }
   bss_end = .;
}
```

现在重新编译烧写，测试结果正常。 再次查看反汇编文件，发现现在 bss 段以 4 字节对齐，清理 bss 段也是正常的。

```
Disassembly of section .bss:

30000004 <g_A>:
30000004:	00000000 	andeq	r0, r0, r0
```

```
30000008 <g_B>:
30000008:	00000000	andeq	r0, r0, r0
Disassembly of section .comment:
```

同样的问题也会出在代码重定位这里，如何保证 data 段起始地址也是向 4 对齐呢？ 也是使用 ALIGN(4)向 4 取整。

```
SECTIONS
{
   . = 0x30000000;

   . = ALIGN(4);
   .text      :
   {
     *(.text)
   }

   . = ALIGN(4);
   .rodata : { *(.rodata) }

   . = ALIGN(4);
   .data : { *(.data) }

   . = ALIGN(4);
   __bss_start = .;
   .bss : { *(.bss) *(.COMMON) }
   _end = .;
}
```

Uboot 是裸机的集大成者，可以参考 uboot 链接脚本也是类似的。

第 005 节_代码重定位与位置无关码

一个程序，由代码段、只读数据段、数据段、bss 段等组成。 程序一开始可以烧在 Nor Flash 上面，运行时代码段仍可以在 Nor Flash 运行，但对于数据段，就必须把数据段移到 SDRAM 中，因为只要在 SDRAM 里面，数据段的变量才能被写操作，把程序从一个位置移动到另一个位置，把这个过程就称为**重定位**。 前面的例子，我们只是重定位了数据段，这里我们再尝试重定位整个代码。

先梳理下把整个程序复制到 SDRAM 需要哪些技术细节：

1. 把程序从 Flash 复制到运行地址，链接脚本中就要指定运行地址为 SDRAM 地址；
2. 编译链接生成的 bin 文件，需要在 SDRAM 地址上运行，但上电后却必须先在 0 地址运行，这就要求重定位之前的代码与位置无关(是位置无关码)；

参考 Uboot 修改链接脚本：

```
SECTIONS
{
    . = 0x30000000;

    . = ALIGN(4);
    .text      :
    {
      *(.text)
    }

    . = ALIGN(4);
    .rodata : { *(.rodata) }

    . = ALIGN(4);
    .data : { *(.data) }

    . = ALIGN(4);
    __bss_start = .;
    .bss : { *(.bss) *(.COMMON) }
    _end = .;
}
```

现在我们写的这个链接脚本，称为**一体式**链接脚本，对比前面的**分体式**链接脚本区别在于代码段和数据段的存放位置是否是分开的。

例如现在的一体式链接脚本的代码段后面依次就是只读数据段、数据段、bss 段，都是连续在一起的。 分体式链接脚本则是代码段、只读数据段，中间相关很远之后才是数据段、bss 段。

我们以后的代码更多的采用**一体式**链接脚本，原因如下：

1. 分体式链接脚本适合单片机，单片机自带有 flash，不需要再将代码复制到内存占用空间。而我们的嵌入式系统内存非常大，没必要节省这点空间，并且有些嵌入式系统没有 Nor Flash 等可以直接运行代码的 Flash，就需要从 Nand Flash 或者 SD 卡复制整个代码到内存；
2. JTAG 等调试器一般只支持一体式链接脚本；

修改 start.S 段

```
    /* 重定位 text, rodata, data 段整个程序 */
    mov r1, #0
    ldr r2, =_start         /* 第 1 条指令运行时的地址 */
```

```
    ldr r3, =__bss_start    /* bss 段的起始地址 */

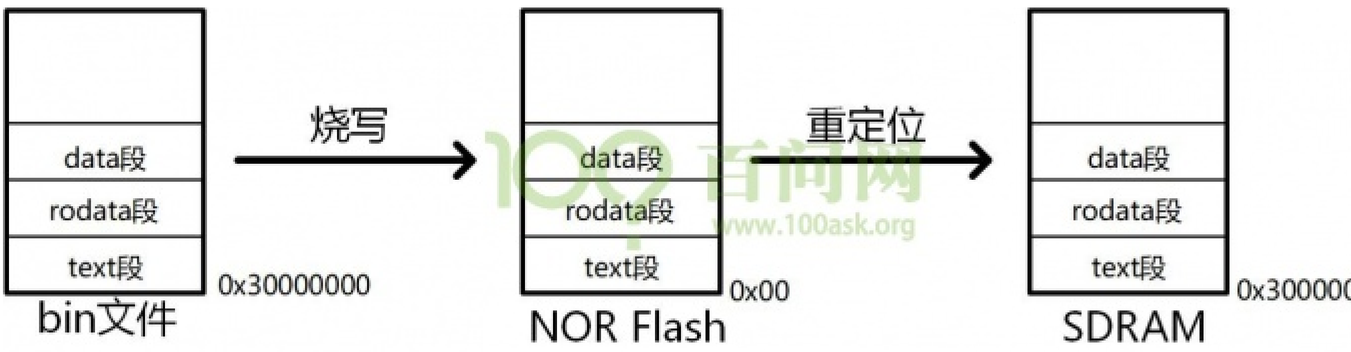
cpy:
    ldr r4, [r1]
    str r4, [r2]
    add r1, r1, #4
    add r2, r2, #4
    cmp r2, r3
    ble cpy


    /* 清除 BSS 段 */
    ldr r1, =__bss_start
    ldr r2, =_end
    mov r3, #0
clean:
    str r3, [r1]
    add r1, r1, #4
    cmp r1, r2
    ble clean

    bl main

halt:
    b halt
```

将修改后的代码重新编译烧写在 Nor Flash 上，上电运行。 对本代码的启动情况进行分析:



在生成的 bin 文件里，代码保存的位置是 0x30000000。随后烧写到 NOR Flash 的 0 地址，但代码的结构没有变化。之后再重定位到 SDRAM。

查看反汇编:

```
    3000005c: eb000106     bl     30000478 <sdram_init>

    30000060: e3a01000     mov    r1, #0 ; 0x0
    30000064: e59f204c     ldr    r2, [pc, #76]; 300000b8
<.text+0xb8>
    30000068: e59f304c     ldr    r3, [pc, #76]; 300000bc
<.text+0xbc>
```

这里的 bl 30000478 不是跳转到 30000478，这个时候 sdram 并未初始化;

为了验证，我们做另一个实验，修改连接脚本 sdram.lds，链接地址改为 0x32000478，编译，查看反汇编：

```
    3000005c: eb000106     bl     30000478 <sdram_init>

    30000060: e3a01000     mov    r1, #0 ; 0x0
    30000064: e59f204c     ldr    r2, [pc, #76]; 300000b8
<.text+0xb8>
    30000068: e59f304c     ldr    r3, [pc, #76]; 300000bc
<.text+0xbc>
```

可以看到现在变成了 bl 30000478,但两个的机器码 eb000106 都是一样的，机器码一样，执行的内容肯定都是一样的。 因此这里并不是跳转到显示的地址，而是跳转到：pc + offset，这个由链接器决定。

假设程序从 0x30000000 执行，当前指令地址：0x3000005c ，那么就是跳到 0x30000478；如果程序从 0 运行，当前指令地址:0x5c 调到：0x00000478

跳转到某个地址并不是由 bl 指令所决定，而是由当前 pc 值决定。反汇编显示这个值只是为了方便读代码。

重点： 反汇编文件里， B 或 BL 某个值，只是起到方便查看的作用，并不是真的跳转。

怎么写位置无关码？

1. 使用相对跳转命令 b 或 bl;

2. 重定位之前，不可使用绝对地址，不可访问全局变量/静态变量，也不可访问有初始值的数组(因为初始值放在 rodata 里，使用绝对地址来访问)；
3. 重定位之后，使用 ldr pc = xxx，跳转到/runtime 地址；

写位置无关码，其实就是不使用绝对地址，判断有没有使用绝对地址，除了前面的几个规则，最根本的办法看反汇编。

因此，前面的例子程序使用 bl 命令相对跳转，程序仍在 NOR/sram 执行，要想让 main 函数在 SDRAM 执行，需要修改代码：

```
//bl main  /*bl 相对跳转，程序仍在 NOR/sram 执行*/
ldr pc, =main/*绝对跳转，跳到 SDRAM*/
```

第 006 节_重定位_清除 BSS 段的 C 函数实现

在前面，我们使用汇编程序来实现了重定位和清 bss 段，本节我们将使用 C 语言，实现重定位和清除 bss 段。

1. 打开 start.S 把原来的汇编代码删除改为调用 C 函数

```
    /* 重定位 text, rodata, data 段整个程序 */
    mov r1, #0
    ldr r2, =_start          /* 第 1 条指令运行时的地址 */
    ldr r3, =__bss_start    /* bss 段的起始地址 */

cpy:
    ldr r4, [r1]
    str r4, [r2]
    add r1, r1, #4
    add r2, r2, #4
    cmp r2, r3
    ble cpy

    /* 清除 BSS 段 */
    ldr r1, =__bss_start
    ldr r2, =_end
    mov r3, #0
clean:
    str r3, [r1]
    add r1, r1, #4
    cmp r1, r2
```

```
    ble clean
```

改为

```
    /* 重定位 text, rodata, data 段整个程序 */
    mov r0, #0
    ldr r1, =_start         /* 第1条指令运行时的地址 */

    ldr r2, =__bss_start    /* bss 段的起始地址 */

    sub r2, r2, r1                  /*长度*/


    bl copy2sdram  /* src, dest, len */

    /* 清除BSS 段 */
    ldr r0, =__bss_start
    ldr r1, =_end

    bl clean_bss  /* start, end */
```

2. 在 init.c 实现如上两个 C 函数

```
void copy2sdram(volatile unsigned int *src, volatile unsigned int *dest, unsigned int len)  /* src, dest, len */
{
    unsigned int i = 0;

    while (i < len)
    {
        *dest++ = *src++;
        i += 4;
    }
}


void clean_bss(volatile unsigned int *start, volatile unsigned int *end)  /* start, end */
{
    while (start <= end)
    {
```

```
            *start++ = 0;
        }
    }
```

汇编中，为 C 语言传入的参数，依次就是 R1、R2、R3。 编译，烧写运行没有问题。

我们假设不想汇编传入参数，而是 C 语言直接取参数。
1. 修改 start.S 跳转到 C 函数不需要任何参数

```
    bl sdram_init
    //bl sdram_init2     /* 用到有初始值的数组, 不是位置无关码 */

    /* 重定位 text, rodata, data 段整个程序 */
    bl copy2sdram

    /* 清除 BSS 段 */
    bl clean_bss
```

2. 修改链接脚本，让__code_start 等于当前地址，也就是这里的 0x30000000

```
SECTIONS
{
    . = 0x30000000;

    __code_start = .; //定义__code_start 地址位当前地址

    . = ALIGN(4);
    .text      :
    {
      *(.text)
    }

    . = ALIGN(4);
    .rodata : { *(.rodata) }

    . = ALIGN(4);
    .data : { *(.data) }
```

```
        . = ALIGN(4);
        __bss_start = .;
        .bss : { *(.bss) *(.COMMON) }
        _end = .;
    }
```

3.修改 init.c 用函数来获取参数

```
    void copy2sdram(void)
    {
        /* 要从 lds 文件中获得 __code_start, __bss_start
         * 然后从 0 地址把数据复制到__code_start
         */

        extern int __code_start, __bss_start;//声明外部变量

        volatile unsigned int *dest = (volatile unsigned int
*)&__code_start;
        volatile unsigned int *end = (volatile unsigned int
*)&__bss_start;
        volatile unsigned int *src = (volatile unsigned int
*)0;

        while (dest < end)
        {
                *dest++ = *src++;
        }
    }


    void clean_bss(void)
    {
        /* 要从 lds 文件中获得 __bss_start, _end
         */
        extern int _end, __bss_start;

        volatile unsigned int *start = (volatile unsigned
int *)&__bss_start;
        volatile unsigned int *end = (volatile unsigned int
*)&_end;
```

```
        while (start <= end)
        {
                *start++ = 0;
        }
    }
```

编译烧写运行 ，没有问题。

总结：C 函数怎么使用 lds 文件总的变量 abc？

1. 在 C 函数中声明改变量为 extern 外部变量类型，比如：`extern int abc;`
2. 使用时，要取址，比如：`int *p = &abc;//p 的只即为 lds 文件中 abc 的值`

汇编文件中可以直接使用外部链接脚本中的变量,但 C 函数中要加上取址符号。

解释一下原因： C 函数中，定义一个全局变量 `int g_i;`，程序中必然有 4 字节的空间留出来给这个变量 `g_i`。

假如我们的 lds 文件中有很多变量 lds{ a1 = ; a2 = ; a3 = ; ... } 如果我们 C 程序只用到几个变量，完全没必要全部存储 lds 里面的所有变量，C 程序是不保存 lds 中的变量的。 对于万一要用到的变量，编译程序时，有一个

symbol table 符号表：



如何使用 symbol table 符号表？

1. 对于常规变量 g_i，得到里面的值，使用&g_i 得到 addr;
2. 为了保持代码的一致，对于 lds 中的 a1，使用&a1 得到里面的值;

这只是一个编译器的小技巧，不用深究。
结论：

1. C 程序中不保存 lds 文件中的变量，lds 再大也不影响;
2. 借助 symbol table 保存 lds 的变量，使用时加上"&"得到它的值，链接脚本的变量要在 C 程序中声明为外部变量，任何类型都可以;

第 014 课 异常与中断

第 001 节_概念引入与处理流程

取个场景解释中断 假设有个大房间里面有小房间，婴儿正在睡觉，他的妈妈在外面看书。 问：这个母亲怎么才能知道这个小孩醒？

1. 过一会打开一次房门，看婴儿是否睡醒，让后接着看书
2. 一直等到婴儿发出声音以后再过去查看，期间都在读书

第一种 叫做查询方式：

- 优点：简单
- 缺点： 累

写程序如何：

```
while(1)
{
    1 read book(读书)

    2 open door（开门）

    if(睡)
           return(read book)
    else
           照顾小孩

}
```

第二种叫中断方式： 优点：不累 缺点：复杂
写程序：

```
while(1)
{
    read book
    中断服务程序()//如何被调用？
    {
    处理照顾小孩
    }
}
```

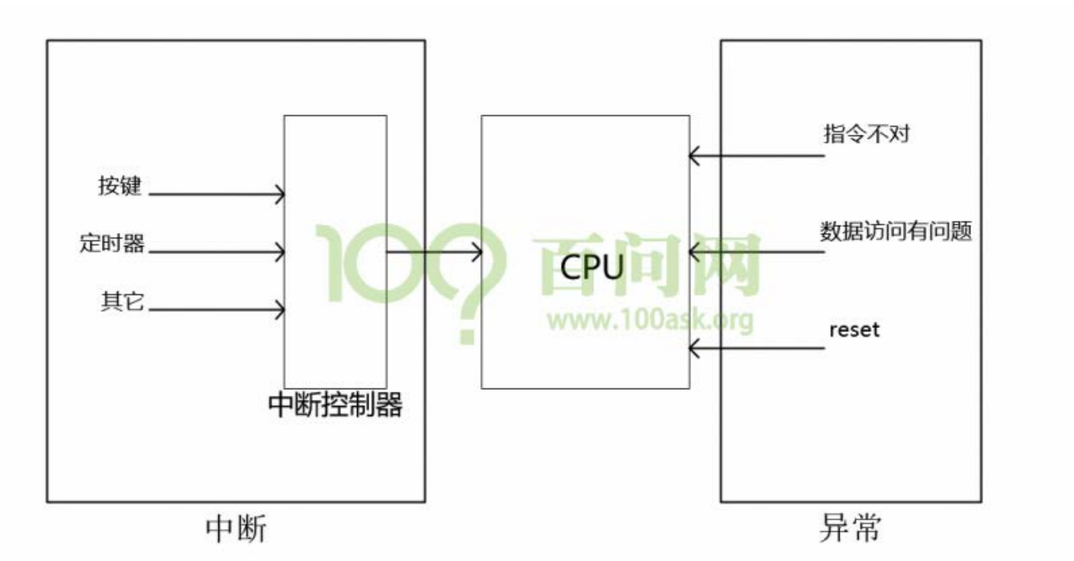
我们还是看看母亲被小孩哭声打断如何照顾小孩？
母亲的处理过程 1 平时看书 2 发生了各种声音，如何处理这些声音 有远处的猫叫（听而不闻，忽略） 门铃声有快递（开门收快递） 小孩哭声（打开

房门，照顾小孩） 3 母亲的处理 只会处理门铃声和小孩哭声 a 现在书中放入书签，合上书(保存现场) b 去处理 (调用对应的中断服务程序) c 继续看书(恢复现场)

不同情况，不同处理 a 对于门铃：开门取快件 b 对于哭声:照顾小孩

我们将母亲的处理过程抽象化 母亲的头脑相当于 CPU 耳朵听到声音会发送信号给脑袋，声音来源有很多种，有远处的猫叫，门铃声，小孩哭声。这些声音传入耳朵，再由耳朵传给大脑，除了这些可以中断母亲的看书，还有其他情况，比如身体不舒服，有只蜘蛛掉下来，对于特殊情况无法回避，必须立即处理

对比我们得 arm 系统



有 CPU 有中断控制器，我们的中断源发送 中断控制器可以发信号给 CPU 告诉它发生了那些紧急情况 中断源有 按键 定时器 有其它的（比如网络数据） 这些信号都可以发送信号给中断控制器，再由中断控制器发送信号给 CPU 表明有这些中断产生了，这些成为中断(属于一种异常)

还有什么可以中断 CPU 运行 指令不对，数据访问有问题 reset 信号，这些都可以中断 CPU 这些成为异常中断

重点在于 保存现场以及恢复现场

处理过程 a 保存现场(各种寄存器) b 处理异常(中断属于一种异常) c 恢复现场

arm 对异常(中断)处理过程 1 初始化: a 设置中断源，让它可以产生中断 b 设置中断控制器(可以屏蔽某个中断，优先级) c 设置 CPU 总开关，(使能中断)

2 执行其他程序:正常程序

3 产生中断:按下按键--->中断控制器--->CPU

4 cpu 每执行完一条指令都会检查有无中断/异常产生 5 发现有中断/异常产生，开始处理 **对于不同的异常，跳去不同的地址**执行程序 这地址上，只是一条跳转指令，跳去执行某个函数 (地址) 指的是异常向量 如下就是异常向量表 对于不同的异常都有一条跳转指令

```
.globl _start
_start: b reset
   ldr    pc, _undefined_instruction
   ldr    pc, _software_interrupt
   ldr    pc, _prefetch_abort
   ldr    pc, _data_abort
   ldr    pc, _not_used
   ldr    pc, _irq //发生中断时，CPU 跳到这个地址执行该指令
**假设地址为 0x18**
   ldr    pc, _fiq
//我们先在 0x18 这里放 ldr pc ,__irq,于是 cpu 最终会跳去执行
__irq 代码
//保护现场，调用处理函数，恢复现场
```

3-5 都是硬件强制做的

6 这些函数做什么事情()软件做的 a 保存现场(各种寄存器) b 处理异常(中断): 分辨中断源 再调用不同的处理函数 c 恢复现场

对比母亲的处理过程来比较 arm 中断的处理过程

中断处理程序怎么被调用? CPU--->0x18 --跳转到其他函数->

做保护现场

调用函数

分辨中断源 调用对应函数

恢复现场

cpu 到 0x18 是由硬件决定的，跳去执行更加复杂函数(由软件决定)

第 002 节_CPU 模式(Mode)_状态(State)与寄存器

这节课我们来讲 CPU 的工作模式(Mode) 状态(State)寄存器 7 种 Mode:

usr/sys

undefined(und)

Supervisor(svc)

Abort(abt)

IRQ(irq)

FIQ(fiq)

2种State:

ARM state
Thumb state

寄存器:

通用寄存器
备份寄存器(banked register)
当前程序状态寄存器(Current Program Status Register);CPSR
CPSR的备份寄存器:SPSR(Save Program Status Register)

我们仍然以这个母亲为例讲解这个CPU模式 这个母亲无压力看书 -->(正常模式) 要考试，看书--->(兴奋模式) 生病---->(异常模式)

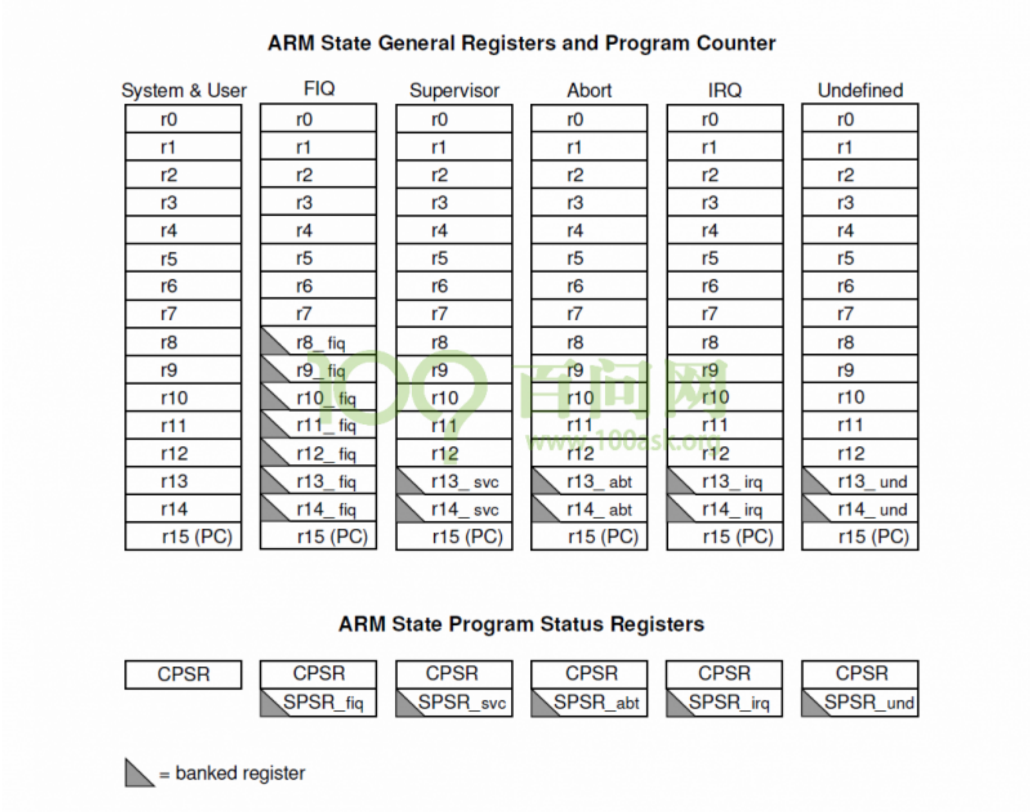
可以参考书籍 《ARM体系结构与编程》作者：杜春雷

对于ARM CPU有7种模式 1 usr ：类比 正常模式 2 sys ：类比的话兴奋模式 3 5种异常模式：(2440用户手册72页) 3.1 und ：未定义模式 3.2 svc ：管理模式 3.3 abt ：终止模式 a 指令预取终止(读写某条错误的指令导致终止运行) b 数据访问终止 (读写某个地址，这个过程出错) 都会进入终止模式 3.4 IRQ： 中断模式 3.5 FIQ： 快中断模式

我们可以称以下6种为特权模式

und ：未定义模式
svc ：管理模式
abt ：终止模式
IRQ ：中断模式
FIQ ：快中断模式
sys ：系统模式

usr 用户模式(不可直接进入其他模式) 可以编程操作 CPSR 直接进入其他模



式

这个图是有关各个模式下能访问寄存器的，再讲这个图之前我们先引入 2 种 state

CPU 有两种 state 1 ARM state 使用 ARM 指令集，每个指令 4byte 2 Thumb state 使用的是 Thumb 指令集，每个指令 2byte

比如同样是 mov R0, R1 编译后 对于 ARM 指令集要占据 4 个字节：机器码 对于 Thumb 指令集占据 2 个字节：机器码 引入 Thumb 减少存储空间

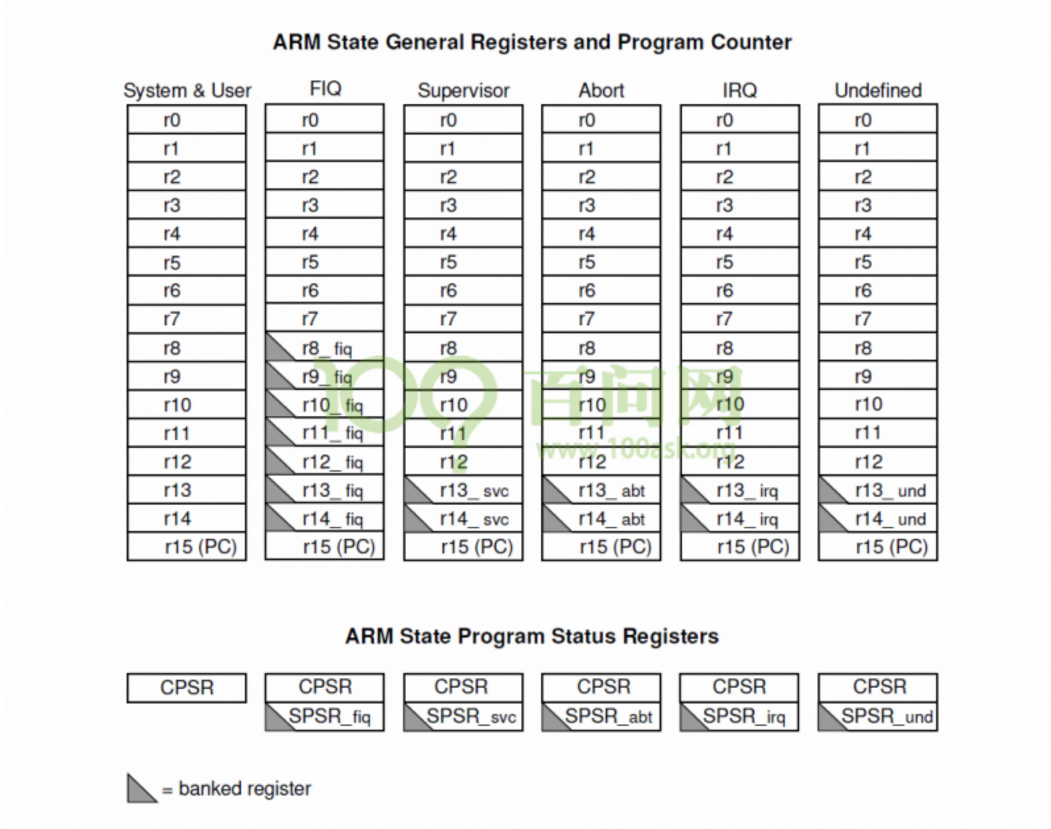
百度搜索 ARM 指令集和 Thumb 指令的区别

现在先区分下 ARM 指令集与 Thumb 指令集

Thumb 指令可以看作是 ARM 指令压缩形式的子集,是针对代码密度的问题而提出的,它具有 16 位的代码密度但是它不如 ARM 指令的效率高 .Thumb 不是一个完整的体系结构,不能指望处理只执行 Thumb 指令而不支持 ARM 指令集. 因此,Thumb 指令只需要支持通用功能,必要时可以借助于完善的 ARM 指令集, 比如,所有异常自动进入 ARM 状态.在编写 Thumb 指令时,先要使用伪指令 CODE16 声明,而且在 ARM 指令中要使用 BX 指令跳转到 Thumb 指令,以切换处理器状态.编写 ARM 指令时,则可使用伪指令 CODE32 声明.

下节课会演示使用 Thumb 指令集编译，看是否生成的 bin 文件会变小很

多



在每种模式下都有 R0 ~ R15 在这张图注意到有些寄存器画有灰色的三角形，表示访问该模式下访问的专属寄存器 比如

```
mov R0, R8
mov R0, R8
```

在 System 模式下访问的是 R0 ~ R8,在所有模式下访问 R0 都是同一个寄存器

```
mov R0,R8_fiq
```

但是在 FIQ 模式下，访问 R8 是访问的 FIQ 模式专属的 R8 寄存器，不是同一个物理上的寄存器

在这五种异常模式中每个模式都有自己专属的 R13 R14 寄存器，R13 用作 SP(栈) R14 用作 LR(返回地址) LR 是用来保存发生异常时的指令地址

为什么快中断(FIQ)有那么多专属寄存器，这些寄存器称为备份寄存器

回顾一下中断的处理过程 1 保存现场(保存被中断模式的寄存器) 就比如说我们的程序正在系统模式/用户模式下运行，当你发生中断时，需要把 R0 ~ R14 这些寄存器全部保存下来，让后处理异常，最后恢复这些寄存器 但如果是快中断，那么我就不需要保存 系统/用户模式下的 R8 ~ R12 这几个寄存器，在 FIQ 模式下有自己专属的 R8 ~ R12 寄存器，省略保存寄存器的时间，加快处理速度 但是在 Linux 中并不会使用 FIQ 模式 2 处理 3 恢复现场

CRSR 当前程序状态寄存器，这是一个特别重要的寄存器 SPSR 保存的程序状态寄存器，他们格式如

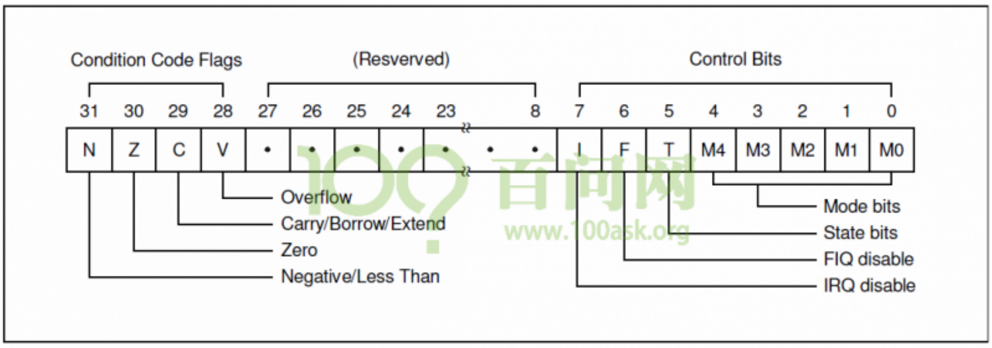


Figure 2-6. Program Status Register Format

下

首先 M4 ~ M0 表示当前 CPU 处于哪一种模式(Mode) 我们可以读取这 5 位来判断 CPU 处于哪一种模式，也可以修改这一种模式位，让其修改这种模式 假如你当前处于用户模式下，是没有权限修改这些位的 M4 ~ M0 对应什么值，会有说明

Table 2-1. PSR Mode Bit Values

M[4:0]	Mode	Visible THUMB state registers	Visible ARM state registers
10000	User	R7..R0, LR, SP PC, CPSR	R14..R0, PC, CPSR
10001	FIQ	R7..R0, LR_fiq, SP_fiq PC, CPSR, SPSR_fiq	R7..R0, R14_fiq..R8_fiq, PC, CPSR, SPSR_fiq
10010	IRQ	R7..R0, LR_irq, SP_irq PC, CPSR, SPSR_irq	R12..R0, R14_irq, R13_irq, PC, CPSR, SPSR_irq
10011	Supervisor	R7..R0, LR_svc, SP_svc, PC, CPSR, SPSR_svc	R12..R0, R14_svc, R13_svc, PC, CPSR, SPSR_svc
10111	Abort	R7..R0, LR_abt, SP_abt, PC, CPSR, SPSR_abt	R12..R0, R14_abt, R13_abt, PC, CPSR, SPSR_abt
11011	Undefined	R7..R0 LR_und, SP_und, PC, CPSR, SPSR_und	R12..R0, R14_und, R13_und, PC, CPSR
11111	System	R7..R0, LR, SP PC, CPSR	R14..R0, PC, CPSR

查看其他位

Bit5 State bits 表示 CPU 工作与 Thumb State 还是 ARM State 用的指令集是什么

Bit6 FIQ disable 当 bit6 等于 1 时，FIQ 是不工作的

Bit7 IRQ disable 当 bit5 等于 1 时，禁止所有的 IRQ 中断，这个位是 IRQ 的总开关

Bit8 ~ Bit27 是保留位

Bite28 ~ Bit31 是状态位，

什么是状态位，比如说执行一条指令

```
cmp R0, R1
```

如果 R0 等于 R1 那么 zero 位等于 1，这条指令影响 Z 位，如果 R0 == R1，则 Z = 1

beq 跳转到 xxx 这条指令会判断 Bit30 是否为 1，是 1 的话则跳转，不是 1 的话则不会跳转 使用 Z 位，如果 Z 位等于 1 则跳转，这些指令是借助状态位实现的

SPSR 保存的程序状态寄存器 表示发生异常时这个寄存器会用来保存被中断的模式下他的 CPSR 就比如我我的程序在系统模式下运行 CPSR 是某个值，当发生中断时会进入 irq 模式，这个 CPSR_irq 就保存系统模式下的 CPSR 我们来看看发生异常时 CPU 是如何协同工作的 进入异常的处理流程(硬

Action on Entering an Exception

While handling an exception, the ARM920T does following activities:

1. Preserves the address of the next instruction in the appropriate Link Register. If the exception has been entered from ARM state, then the address of the next instruction is copied into the Link Register (that is, current PC + 4 or PC + 8 depending on the exception. See Table 2-2 on for details). If the exception has been entered from THUMB state, then the value written into the Link Register is the current PC offset by a value such that the program resumes from the correct place on return from the exception. This means that the exception handler need not determine which state the exception was entered from. For example, in the case of SWI, MOVS PC, R14_svc will always return to the next instruction regardless of whether the SWI was executed in ARM or THUMB state.
2. Copies the CPSR into the appropriate SPSR
3. Forces the CPSR mode bits to a value which depends on the exception
4. Forces the PC to fetch the next instruction from the relevant exception vector

件)

我们来翻译一下： 发生异常时，我们的 CPU 会做什么事情 1 把下一条指令的地址保存在 LR 寄存器里(某种异常模式的 LR 等于被中断的下一条指令的地址) 它有可能是 PC + 4 有可能是 PC + 8,到底是那种取决于不同的情况 2 把 CPSR 保存在 SPSR 里面(某一种异常模式下 SPSR 里面的值等于 CPSR) 3 修改 CPSR 的模式为进入异常模式(修改 CPSR 的 M4 ~ M0 进入异常模式) 4 跳到向量表

退出异常怎么做？

Action on Leaving an Exception

On completion, the exception handler:

1. Moves the Link Register, minus an offset where appropriate, to the PC. (The offset will vary depending on the type of exception.)
2. Copies the SPSR back to the CPSR
3. Clears the interrupt disable flags, if they were set on entry

1 让 LR 减去某个值，让后赋值给 PC(PC = 某个异常 LR 寄存器减去 offset) 减去什么值呢？ 也就是我们怎么返回去继续执行原来的程序，根据下

面这个表来取值

Table 2-2. Exception Entry/Exit

Return Instruction		Previous State		Notes
		ARM R14_x	THUMB R14_x	
BL	MOV PC, R14	PC + 4	PC + 2	(1)
SWI	MOVS PC, R14_svc	PC + 4	PC + 2	(1)
UDEF	MOVS PC, R14_und	PC + 4	PC + 2	(1)
FIQ	SUBS PC, R14_fiq, #4	PC + 4	PC + 4	(2)
IRQ	SUBS PC, R14_irq, #4	PC + 4	PC + 4	(2)
PABT	SUBS PC, R14_abt, #4	PC + 4	PC + 4	(1)
DABT	SUBS PC, R14_abt, #8	PC + 8	PC + 8	(3)
RESET	NA	–	–	(4)

如果发生的是 SWI 可以把 R14_svc 复制给 PC 如果发生的是 IRQ 可以把 R14_irq 的值减去 4 赋值给 PC 2 把 CPSR 的值恢复(CPSR 值等于 某一个一场模式下的 SPSR) 3 清中断（如果是中断的话，对于其他异常不用设置）

第 003 节_不重要_Thumb 指令集程序示例

在上节视频里说 ARMCPU 有两种状态 ARM State 每条指令会占据 4byte Thumb State 每条指令占据 2byte

我们说过 Thumb 指令集并不重要，本节演示把一个程序使用 Thumb 指令集来编译它 使用上一章节的重定位代码 打开 Makefile 和 Start.S

Makefile 文件

```
all:
	arm-linux-gcc -c -o led.o led.c
	arm-linux-gcc -c -o uart.o uart.c
	arm-linux-gcc -c -o init.o init.c
	arm-linux-gcc -c -o main.o main.c
	arm-linux-gcc -c -o start.o start.S
	#arm-linux-ld -Ttext 0 -Tdata 0x30000000  start.o led.o uart.o init.o main.o -o sdram.elf
	arm-linux-ld -T sdram.lds start.o led.o uart.o init.o main.o -o sdram.elf
	arm-linux-objcopy -O binary -S sdram.elf sdram.bin
	arm-linux-objdump -D sdram.elf > sdram.dis
clean:
	rm *.bin *.o *.elf *.dis
```

对于使用 Thumb 指令集

```
all:
```

```
        arm-linux-gcc -mthumb -c -o led.o led.c//只需要在arm-linux-gcc 加上 mthumb 命令即可
        arm-linux-gcc -c -o uart.o uart.c
        arm-linux-gcc -c -o init.o init.c
        arm-linux-gcc -c -o main.o main.c
        arm-linux-gcc -c -o start.o start.S
        #arm-linux-ld -Ttext 0 -Tdata 0x30000000  start.o led.o uart.o init.o main.o -o sdram.elf
        arm-linux-ld -T sdram.lds start.o led.o uart.o init.o main.o -o sdram.elf
        arm-linux-objcopy -O binary -S sdram.elf sdram.bin
        arm-linux-objdump -D sdram.elf > sdram.dis
    clean:
        rm *.bin *.o *.elf *.dis
```

改进

```
    all: led.o uart.o init.o main.o start.o //all 依赖 led.o uart.o init.o main.o start.o
        #arm-linux-ld -Ttext 0 -Tdata 0x30000000  start.o led.o uart.o init.o main.o -o sdram.elf
        arm-linux-ld -T sdram.lds start.o led.o uart.o init.o main.o -o sdram.elf
        arm-linux-objcopy -O binary -S sdram.elf sdram.bin
        arm-linux-objdump -D sdram.elf > sdram.dis
    clean:
        rm *.bin *.o *.elf *.dis

    %.o : %.c
        arm-linux-gcc -mthumb -c -o $@ $< //对于所有的.c 文件使用规则就可以使用 thumb 指令集编译 $@表示目标 $<表示第一个依赖

    %.o : %.S
        arm-linux-gcc -c -o $@ $<
```

对 start.S 需要修改代码

原重定位章节 Start.S 文件

```
    .text
    .global _start

    _start:

        /* 关闭看门狗 */

        ldr r0, =0x53000000
        ldr r1, =0
        str r1, [r0]


        /* 设置MPLL, FCLK : HCLK : PCLK = 400m : 100m : 50m
*/
        /* LOCKTIME(0x4C000000) = 0xFFFFFFFF */
        ldr r0, =0x4C000000
        ldr r1, =0xFFFFFFFF
        str r1, [r0]

        /* CLKDIVN(0x4C000014) = 0X5, tFCLK:tHCLK:tPCLK =
1:4:8  */
        ldr r0, =0x4C000014
        ldr r1, =0x5
        str r1, [r0]


        /* 设置CPU工作于异步模式 */

        mrc p15,0,r0,c1,c0,0
        orr r0,r0,#0xc0000000   //R1_nF:OR:R1_iA
        mcr p15,0,r0,c1,c0,0


        /* 设置MPLLCON(0x4C000004) = (92<<12)|(1<<4)|(1<<0)
         *  m = MDIV+8 = 92+8=100
         *  p = PDIV+2 = 1+2 = 3
         *  s = SDIV = 1
         *  FCLK = 2*m*Fin/(p*2^s) = 2*100*12/(3*2^1)=400M
         */
        ldr r0, =0x4C000004
        ldr r1, =(92<<12)|(1<<4)|(1<<0)
        str r1, [r0]


        /* 一旦设置PLL, 就会锁定lock time直到PLL输出稳定
```

```
 * 然后CPU工作于新的频率FCLK
 */


/* 设置内存: sp 栈 */
/* 分辨是nor/nand启动
 * 写0到0地址, 再读出来
 * 如果得到0, 表示0地址上的内容被修改了, 它对应ram, 这就
是nand启动
 * 否则就是nor启动
 */
mov r1, #0
ldr r0, [r1] /* 读出原来的值备份 */
str r1, [r1] /* 0->[0] */
ldr r2, [r1] /* r2=[0] */
cmp r1, r2   /* r1==r2? 如果相等表示是NAND启动 */
ldr sp, =0x40000000+4096 /* 先假设是nor启动 */
moveq sp, #4096  /* nand启动 */
streq r0, [r1]   /* 恢复原来的值 */

bl sdram_init
//bl sdram_init2    /* 用到有初始值的数组, 不是位置无关码
*/

/* 重定位text, rodata, data段整个程序 */
bl copy2sdram

/* 清除BSS段 */
```

```
	bl clean_bss

	//bl main  /* 使用BL命令相对跳转，程序仍然在NOR/sram执行 */

	ldr pc, =main  /* 绝对跳转，跳到SDRAM */

halt:
	b halt
```

使用 thumb 指令集的 Start.S 文件

```
.text
.global _start

.code 32 //表示后续的指令使用ARM指令集

_start:

	/* 关闭看门狗 */
	ldr r0, =0x53000000
	ldr r1, =0
	str r1, [r0]

	/* 设置MPLL, FCLK : HCLK : PCLK = 400m : 100m : 50m */
	/* LOCKTIME(0x4C000000) = 0xFFFFFFFF */
	ldr r0, =0x4C000000
	ldr r1, =0xFFFFFFFF
	str r1, [r0]

	/* CLKDIVN(0x4C000014) = 0X5, tFCLK:tHCLK:tPCLK = 1:4:8  */
	ldr r0, =0x4C000014
	ldr r1, =0x5
	str r1, [r0]

	/* 设置CPU工作于异步模式 */
	mrc p15,0,r0,c1,c0,0
```

```
orr r0,r0,#0xc0000000   //R1_nF:OR:R1_iA
mcr p15,0,r0,c1,c0,0

/* 设置MPLLCON(0x4C000004) = (92<<12)|(1<<4)|(1<<0)
 *  m = MDIV+8 = 92+8=100
 *  p = PDIV+2 = 1+2 = 3
 *  s = SDIV = 1
 *  FCLK = 2*m*Fin/(p*2^s) = 2*100*12/(3*2^1)=400M
 */
ldr r0, =0x4C000004
ldr r1, =(92<<12)|(1<<4)|(1<<0)
str r1, [r0]

/* 一旦设置PLL, 就会锁定lock time直到PLL输出稳定
 * 然后CPU工作于新的频率FCLK
 */



/* 设置内存: sp 栈 */
/* 分辨是nor/nand启动
 * 写0到0地址, 再读出来
 * 如果得到0, 表示0地址上的内容被修改了, 它对应ram, 这就
是nand启动
 * 否则就是nor启动
 */
mov r1, #0
ldr r0, [r1] /* 读出原来的值备份 */
str r1, [r1] /* 0->[0] */
ldr r2, [r1] /* r2=[0] */
cmp r1, r2   /* r1==r2? 如果相等表示是NAND启动 */
ldr sp, =0x40000000+4096 /* 先假设是nor启动 */
```

```
        moveq sp, #4096  /* nand 启动 */

        streq r0, [r1]   /* 恢复原来的值 */


        /* 怎么从ARM State 切换到Thumb State? */

        adr r0, thumb_func //定义此标号的地址

        add r0, r0, #1  /* bit0=1 时, bx 就会切换CPU State 到
thumb state */
        bx r0

    .code 16 //下面都使用thumb 指令集

    thumb_func:      //需要得到这个标号的地址

        /*下面就是使用thumb 指令来执行程序*/

        bl sdram_init

        //bl sdram_init2    /* 用到有初始值的数组, 不是位置无关码
*/


        /* 重定位text, rodata, data 段整个程序 */

        bl copy2sdram


        /* 清除BSS 段 */

        bl clean_bss


        //bl main  /* 使用BL 命令相对跳转, 程序仍然在NOR/sram 执
行 */

        ldr r0, =main  /* 绝对跳转, 跳到SDRAM ,先把main 的地址赋
值给R0 */

        mov pc, r0  /*让后再移动到PC*/
```

```
halt:
    b halt
```

上传代码编译测试 出现错误,如下 init.o(.text+0x6c):In function 'sdram_init2'; undefined reference to 'memcpy' 发现是 init,o 里 sdram——init2 使用的了 memcpy 函数

查看 init.c

```
#include "s3c2440_soc.h"

void sdram_init(void)
{
    BWSCON = 0x22000000;

    BANKCON6 = 0x18001;
    BANKCON7 = 0x18001;

    REFRESH  = 0x8404f5;

    BANKSIZE = 0xb1;

    MRSRB6   = 0x20;
    MRSRB7   = 0x20;
}

#if 0

/*********************************************************************************
 * 设置控制SDRAM 的13 个寄存器
 * 使用位置无关代码
 ********************************************************************************/
void memsetup(void)
{
    unsigned long *p = (unsigned long *)MEM_CTL_BASE;
    p[0] = 0x22111110;        //BWSCON
    p[1] = 0x00000700;        //BANKCON0
    p[2] = 0x00000700;        //BANKCON1
```

```
    p[3] = 0x00000700;       //BANKCON2
    p[4] = 0x00000700;       //BANKCON3
    p[5] = 0x00000700;       //BANKCON4
    p[6] = 0x00000700;       //BANKCON5
    p[7] = 0x00018005;       //BANKCON6
    p[8] = 0x00018005;       //BANKCON7
    p[9] = 0x008e07a3;
    //REFRESH,HCLK=12MHz:0x008e07a3,HCLK=100MHz:0x008e0
4f4
    p[10] = 0x000000b2;      //BANKSIZE
    p[11] = 0x00000030;      //MRSRB6
    p[12] = 0x00000030;      //MRSRB7
}


#endif
```

/*下面函数使用了 memcpy 函数，显然是编译器的操作，使用了 memcpy 把数组里的值从代码段拷贝到了 arr 局部变量里 是否可以禁用掉 memcpy*/

```
void sdram_init2(void)
{
    unsigned int arr[] = {
            0x22000000,  //BWSCON
            0x00000700,  //BANKCON0
            0x00000700,  //BANKCON1
            0x00000700,  //BANKCON2
            0x00000700,  //BANKCON3
            0x00000700,  //BANKCON4
            0x00000700,  //BANKCON5
            0x18001,     //BANKCON6
            0x18001,     //BANKCON7
            0x8404f5,
    //REFRESH,HCLK=12MHz:0x008e07a3,HCLK=100MHz:0x008e0
4f4
             0xb1, //BANKSIZE
             0x20, //MRSRB6
             0x20, //MRSRB7

            };
    volatile unsigned int * p = (volatile unsigned int
*)0x48000000;
    int i;

    for (i = 0; i < 13; i++)
```

```
        {
                *p = arr[i];
                p++;
        }

    }
```

文章说没有什么方法禁用 memecpy 但是可以修改这些变量

比如说将其修改为静态变量，这些数据就会放在数据段中，最终重定位时会把数据类拷贝到对应的 arr 地址里面去

```
    void sdram_init2(void)
    {

        const static unsigned int arr[] = {  //加上 const 和
static
                0x22000000,  //BWSCON
                0x00000700,  //BANKCON0
                0x00000700,  //BANKCON1
                0x00000700,  //BANKCON2
                0x00000700,  //BANKCON3
                0x00000700,  //BANKCON4
                0x00000700,  //BANKCON5
                0x18001,     //BANKCON6
                0x18001,     //BANKCON7
                0x8404f5,
        //REFRESH,HCLK=12MHz:0x008e07a3,HCLK=100MHz:0x008e0
4f4
                 0xb1, //BANKSIZE
                 0x20, //MRSRB6
                 0x20, //MRSRB7

                };
        volatile unsigned int * p = (volatile unsigned int
*)0x48000000;
        int i;

        for (i = 0; i < 13; i++)
        {
                *p = arr[i];
                p++;
        }

    }
```

拷贝进行实验
得出 bin 文件有 1.4k 左右



查看之前的文件使用 ARM 指令集是 2K 左右



查看反汇编代码

```
sdram.elf:     file format elf32-littlearm

Disassembly of section .text:

/*前面这些 ARM 指令还是占用 4 个字节*/
30000000 <_start>:
30000000: e3a00453      mov    r0, #1392508928     ;
0x53000000
30000004: e3a01000      mov    r1, #0 ; 0x0
30000008: e5801000      str    r1, [r0]
3000000c: e3a00313      mov    r0, #1275068416     ;
0x4c000000
30000010: e3e01000      mvn    r1, #0 ; 0x0
30000014: e5801000      str    r1, [r0]
30000018: e59f005c      ldr    r0, [pc, #92]; 3000007c
<.text+0x7c>
3000001c: e3a01005      mov    r1, #5 ; 0x5
30000020: e5801000      str    r1, [r0]
30000024: ee110f10      mrc    15, 0, r0, cr1, cr0, {0}
```

```
    30000028: e3800103     orr    r0, r0, #-1073741824
       ; 0xc0000000
    3000002c: ee010f10     mcr    15, 0, r0, cr1, cr0, {0}
    30000030: e59f0048     ldr    r0, [pc, #72]; 30000080
<.text+0x80>
    30000034: e59f1048     ldr    r1, [pc, #72]; 30000084
<.text+0x84>
    30000038: e5801000     str    r1, [r0]
    3000003c: e3a01000     mov    r1, #0 ; 0x0
    30000040: e5910000     ldr    r0, [r1]
    30000044: e5811000     str    r1, [r1]
    30000048: e5912000     ldr    r2, [r1]
    3000004c: e1510002     cmp    r1, r2
    30000050: e59fd030     ldr    sp, [pc, #48]; 30000088
<.text+0x88>
    30000054: 03a0da01     moveq  sp, #4096     ; 0x1000
    30000058: 05810000     streq  r0, [r1]
    3000005c: e28f0004     add    r0, pc, #4    ; 0x4
    30000060: e2800001     add    r0, r0, #1    ; 0x1
    30000064: e12fff10     bx     r0

    30000068 <thumb_func>:
    30000068: f94ef000     bl     30000308 <sdram_init>
    3000006c: f9fef000     bl     3000046c <copy2sdram>
    30000070: fa24f000     bl     300004bc <clean_bss>

    /**下面的 thumb 指令占据 2 个字节**/

    30000074: 4805         ldr    r0, [pc, #20](3000008c
<.text+0x8c>)
    30000076: 4687         mov    pc, r0

    30000078 <halt>:
    30000078: e7fe         b      30000078 <halt>
    3000007a: 0000         lsl    r0, r0, #0
    3000007c: 0014         lsl    r4, r2, #0
    3000007e: 4c00         ldr    r4, [pc, #0] (30000080
<.text+0x80>)
    30000080: 0004         lsl    r4, r0, #0
    30000082: 4c00         ldr    r4, [pc, #0] (30000084
<.text+0x84>)
    30000084: c011         stmia  r0!,{r0, r4}
    30000086: 0005         lsl    r5, r0, #0
    30000088: 1000         asr    r0, r0, #0
    3000008a: 4000         and    r0, r0
```

```
3000008c:04fd          lsl    r5, r7, #19
3000008e:3000          add    r0, #0
```

如果你的 flash 很小的话可以考虑使用 Thumb 指令集

烧写进去看是否可以运行 测试结果没有任何问题 Thumb 指令集后面没有任何作用，只是简单作为介绍

第 004 节_und 异常模示程序示例

写一个程序故意让其发生未定义异常，让后处理这个异常 查看 uboot 中源码 uboot\u-boot-1.1.6\cpu\arm920t 打开 start.S

```
/*code: 28 -- 72*/
#include <config.h>
#include <version.h>


/*
*****************************************************************************
 *
 * Jump vector table as in table 3.1 in [1]
 *
*****************************************************************************
 */
#define GSTATUS2   (0x560000B4)
#define GSTATUS3   (0x560000B8)
#define GSTATUS4   (0x560000BC)

#define REFRESH(0x48000024)
#define MISCCR (0x56000080)

#define LOCKTIME   0x4C000000   /* R/W, PLL lock time count register */
#define MPLLCON         0x4C000004   /* R/W, MPLL configuration register */
#define UPLLCON         0x4C000008   /* R/W, UPLL configuration register */
#define CLKCON          0x4C00000C   /* R/W, Clock generator control reg. */
```

```
        #define CLKSLOW            0x4C000010   /* R/W, Slow
clock control register */
        #define CLKDIVN            0x4C000014   /* R/W, Clock
divider control */

        /******下面这些就是异常向量表*****/
        .globl _start
        _start:      b   reset
              ldr     pc, _undefined_instruction
              ldr     pc, _software_interrupt
              ldr     pc, _prefetch_abort
              ldr     pc, _data_abort
              ldr     pc, _not_used
              ldr     pc, _irq
              ldr     pc, _fiq

        _undefined_instruction:   .word
undefined_instruction
        _software_interrupt:      .word software_interrupt
        _prefetch_abort:    .word prefetch_abort
        _data_abort:        .word data_abort
        _not_used:          .word not_used
        _irq:               .word irq
        _fiq:               .word fiq

              .balignl 16,0xdeadbeef
```

手册异常向量表定义

Table 2-3. Exception Vectors

Address	Exception	Mode in Entry
0x00000000	Reset	Supervisor
0x00000004	Undefined instruction	Undefined
0x00000008	Software Interrupt	Supervisor
0x0000000C	Abort (prefetch)	Abort
0x00000010	Abort (data)	Abort
0x00000014	Reserved	Reserved
0x00000018	IRQ	IRQ
0x0000001C	FIQ	FIQ

接下来我们写程序

```
        .text
```

```
.global _start
_start:
        b reset  /* vector 0 : reset */  //一上电复位，
是从 0 地址开始执行，跳到 reset 处
        b do_und /* vector 4 : und */ //如果发生未定义指
令异常，就会跳到 0x04 地址未定义指令异常处，执行 do_und 程序

/*假设一上电从 0 地址开始执行，reset，做一系列初始化之后
*故意加入一条未定义指令
und_code:
        .word 0xdeadc0de  /* 未定义指令 */
当 CPU 发现无法执行此条指令时，就会发生未定义指令异常，就会执
行 do_und
        bl print2,
*/
do_und:
        /* 执行到这里之前:
         * 1.  lr_und 保存有被中断模式中的下一条即将执行的指
令的地址
         * 2.  SPSR_und 保存有被中断模式的 CPSR
         * 3.  CPSR 中的 M4-M0 被设置为 11011, 进入到 und 模
式
         * 4.  跳到 0x4 的地方执行程序
         */
   //需要从新设置 sp 栈，指向某一块没有使用的地址
        /* sp_und 未设置, 先设置它 */
```

```
ldr sp, =0x34000000

/* 在und异常处理函数中有可能会修改r0-r12, 所以先保存 */

/* 发生异常时，当前被中断的地址会保存在lr寄存器中 先减后存*/

/* lr是异常处理完后的返回地址, 也要保存 */
stmdb sp!, {r0-r12, lr}

/* 保存现场 */

/* 处理und异常 */

mrs r0, cpsr//把cpsr的值读入r0

ldr r1, =und_string//把下面的字符串地址赋值给r1

bl printException

/* 这些寄存器保存在栈中，把他读取出来就可以了*/

/* 恢复现场 */

/* 先读后加*/

/* 把r0 ~ r12的值从栈中都取出来，并且把原来保存的lr值，赋值到pc中去*/

ldmia sp!, {r0-r12, pc}^  /* ^会把spsr的值恢复到cpsr里 */
```

/*

- 如何定义字符串，可以百度搜索 arm-linux-gcc 汇编 定义字符串
-

- 官方的说明文档
- http://web.mit.edu/gnu/doc/html/as_7.html

.string "str"

Copy the characters in str to the object file. You may specify more than one string to copy, separated by commas. Unless otherwise specified for a particular machine, the assembler marks the end of each string with a 0 byte. You can use any of the escape sequences described in section Strings. 我们使用.str 会自动加上结束符

- /

```
und_string:
        .string "undefined instruction exception"


reset:
        /* 关闭看门狗 */

        ldr r0, =0x53000000
        ldr r1, =0
        str r1, [r0]


        /* 设置 MPLL, FCLK : HCLK : PCLK = 400m : 
100m : 50m */
        /* LOCKTIME(0x4C000000) = 0xFFFFFFFF */
        ldr r0, =0x4C000000
        ldr r1, =0xFFFFFFFF
        str r1, [r0]

        /* CLKDIVN(0x4C000014) = 0X5, 
tFCLK:tHCLK:tPCLK = 1:4:8  */
        ldr r0, =0x4C000014
        ldr r1, =0x5
        str r1, [r0]


        /* 设置 CPU 工作于异步模式 */

        mrc p15,0,r0,c1,c0,0
        orr r0,r0,#0xc0000000   //R1_nF:OR:R1_iA
        mcr p15,0,r0,c1,c0,0
```

```
            /* 设置MPLLCON(0x4C000004) =
(92<<12)|(1<<4)|(1<<0)
             *  m = MDIV+8 = 92+8=100
             *  p = PDIV+2 = 1+2 = 3
             *  s = SDIV = 1
             *  FCLK = 2*m*Fin/(p*2^s) =
2*100*12/(3*2^1)=400M
             */
            ldr r0, =0x4C000004
            ldr r1, =(92<<12)|(1<<4)|(1<<0)
            str r1, [r0]

            /* 一旦设置PLL, 就会锁定lock time直到PLL输出稳
定
             * 然后CPU工作于新的频率FCLK
             */



            /* 设置内存: sp 栈 */
            /* 分辨是nor/nand启动
             * 写0到0地址, 再读出来
             * 如果得到0, 表示0地址上的内容被修改了, 它对应
ram, 这就是nand启动
             * 否则就是nor启动
             */
            mov r1, #0
            ldr r0, [r1] /* 读出原来的值备份 */
            str r1, [r1] /* 0->[0] */
            ldr r2, [r1] /* r2=[0] */
            cmp r1, r2   /* r1==r2? 如果相等表示是NAND启动
*/
            ldr sp, =0x40000000+4096 /* 先假设是nor启动 */
```

```
        moveq sp, #4096  /* nand 启动 */

        streq r0, [r1]   /* 恢复原来的值 */

        bl sdram_init
        //bl sdram_init2    /* 用到有初始值的数组, 不是位置无关码 */

        /* 重定位 text, rodata, data 段整个程序 */
        bl copy2sdram

        /* 清除 BSS 段 */
        bl clean_bss

        bl uart0_init

        bl print1
        /* 故意加入一条未定义指令 */
und_code:
        .word 0xff123456  /* 未定义指令 */
        bl print2

        //bl main  /* 使用 BL 命令相对跳转, 程序仍然在 NOR/sram 执行 */

        ldr pc, =main  /* 绝对跳转, 跳到 SDRAM */

halt:
        b halt
```

如何处理这个异常呢？ 直接 print 打印一句话，新建一个 exception.c 文件

```
#include "uart.h"
```

```
void printException(unsigned int cpsr, char *str)
//cpsr 打印相应的寄存器，str 打印一个字符串
{
    puts("Exception! cpsr = ");\\打印 cpsr

    printHex(cpsr);//输出 cpsr 的值

    puts(" ");//输出空格

    puts(str);//输出 str 值

    puts("\n\r");//回车，换行
}
```

我们打开之前编译过的程序的反汇编文件 里面一定包含了保存恢复

```
30000084 <delay>:
30000084:    e1a0c00d    mov    ip, sp
30000088:    e92dd800    stmdb  sp!, {fp, ip, lr,
pc} //保存 d 是减 b 是存
3000008c:    e24cb004    sub    fp, ip, #4    ; 0x4
30000090:    e24dd004    sub    sp, sp, #4    ; 0x4
30000094:    e50b0010    str    r0, [fp, #-16]
30000098:    e51b3010    ldr    r3, [fp, #-16]
3000009c:    e2433001    sub    r3, r3, #1    ; 0x1
300000a0:    e50b3010    str    r3, [fp, #-16]
300000a4:    e51b3010    ldr    r3, [fp, #-16]
300000a8:    e3730001    cmn    r3, #1 ; 0x1
300000ac:    0a000000    beq    300000b4
<delay+0x30>
300000b0:    eafffff8    b      30000098
<delay+0x14>
300000b4:    e89da808    ldmia  sp, {r3, fp, sp,
pc}//恢复，先读后加
```

上传编译
修改 makefile 添加文件

```
all: start.o led.o uart.o init.o main.o exception.o
    #arm-linux-ld -Ttext 0 -Tdata 0x30000000
start.o led.o uart.o init.o main.o -o sdram.elf
```

```
        arm-linux-ld -T sdram.lds $^ -o sdram.elf
        #用$ ^来包含所有的依赖
        arm-linux-objcopy -O binary -S sdram.elf sdram.bin
        arm-linux-objdump -D sdram.elf > sdram.dis
clean:
        rm *.bin *.o *.elf *.dis

%.o : %.c
        arm-linux-gcc -c -o $@ $<

%.o : %.S
        arm-linux-gcc -c -o $@ $<
        *.dis
```

编译成功烧写 没有输出我们想要的字符串 很多同学想学会如何调试程序这里我们演示

```
sdram:


        bl print1 //添加print1

        /* 故意加入一条未定义指令 */
und_code:
        .word 0xdeadc0de  /* 未定义指令 */

        bl print2 //添加print2,实现这两个函数, 来打印


        //bl main  /* 使用BL 命令相对跳转, 程序仍然在 NOR/sram 执行 */

        ldr pc, =main  /* 绝对跳转, 跳到SDRAM */

halt:
        b halt
```

实现 print1 print2 这两个打印函数，在 uart.c 这个文件里

```c
void print1(void)
{
	puts("abc\n\r");
}

void print2(void)
{
	puts("123\n\r");
}
```

上传代码烧写，发现 print1 print2 并未执行成功

发现在 start.S 并未初始化 uart0_init()，删除 main.c 中的 uart0_init()初始化函数

```
	ldr pc, =sdram
sdram:
	bl uart0_init

	bl print1
	/* 故意加入一条未定义指令 */
und_code:
	.word 0xff123456  /* 未定义指令 */
	bl print2

	//bl main  /* 使用 BL 命令相对跳转, 程序仍然在 NOR/sram 执行 */
	ldr pc, =main  /* 绝对跳转, 跳到 SDRAM */

halt:
	b halt
```

加上 uart0_init，再次编译烧写 程序正常运行，print1 print2 全部打印，表明未定义指令并未运行，难道这个地址是一个已经定义的地址

打开 2440 芯片手册 找到 ARM 指令集

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31	
Cond 0 0 I Opcode S Rn Rd Operand2	Data/Processing/ PSR Transfer
Cond 0 0 0 0 0 0 A S Rd Rn Rs 1 0 0 1 Rm	Multiply
Cond 0 0 0 0 1 U A S RdHi RdLo Rn 1 0 0 1 Rm	Multiply Long
Cond 0 0 0 1 0 B 0 0 Rn Rd 0 0 0 0 1 0 0 1 Rm	Single Data Swap
Cond 0 0 0 1 0 0 1 0 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 1 Rn	Branch and Exchange
Cond 0 0 0 P U 0 W L Rn Rd 0 0 0 0 1 S H 1 Rm	Halfword Data Transfer: register offset
Cond 0 0 0 P U 1 W L Rn Rd Offset 1 S H 1 Offset	Halfword Data Transfer: immendiate offset
Cond 0 1 I P U B W L Rn Rd Offset	Single Data Transfer
Cond 0 1 1 1	Undefined
Cond 1 0 0 P U B W L Rn Register List	Block Data Transfer
Cond 1 0 1 L Offset	Branch
Cond 1 1 0 P U B W L Rn CRd CP# Offset	Coprocessor Data Transfer
Cond 1 1 1 0 CP Opc CRn CRd CP# CP 0 CRm	Coprocessor Data Operation
Cond 1 1 1 0 CP Opc L CRn Rd CP# CP 1 CRm	Coprocessor Register Transfer
Cond 1 1 1 1 Ignored by processor	Software Interrupt
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31	

发现竟然是 SWI 指令，CPU 可以识别出来，他不是一条未定义指令 我们得找到一条 CPU 不能识别的指令,定义为 0x03000000

```
        ldr pc, =sdram
sdram:
        bl uart0_init

        bl print1
        /* 故意加入一条未定义指令 */
und_code:
        .word 0x03000000  /* 未定义指令 */
        bl print2

        //bl main  /* 使用BL命令相对跳转, 程序仍然在
NOR/sram执行 */

        ldr pc, =main  /* 绝对跳转, 跳到SDRAM */

halt:
```

```
        b halt
```

编译烧写执行

打印了未定义指令异常 CPSR 地址,打印了字符串，最后执行 main 函数 .word 0xdeadcode /* 也是一条未定义指令 只要指令地址对不上上表就是未定义指令*/

我们查看下 cpsr 是否处于未定义模式

bit[4:0]表示 CPU 模式 11011 果然处于 und 模式

我们看看这个程序做了什么事情

```
    .text
    .global _start
```

*/*一上电复位，从 0 地址开始执行*

跳到 reset:

做了一系列初始化

当执行到0xdeadc0de 这条指令时候，CPU 根本就不知道这条指令什么意思

```
    und_code:
        .word 0xdeadc0de  /* 未定义指令 */
        bl print2
```

让后就发生未定义指令异常，他会把下一条指令的地址保存到异常模式的 LR 寄存器

```
        /* 执行到这里之前已经发生了很多事情
         * 1. lr_und 保存有被中断模式中的下一条即将执行的指令的地址
         * 2. SPSR_und 保存有被中断模式的 CPSR
         * 3. CPSR 中的 M4-M0 被设置为 11011, 进入到 und 模式
         * 4. 跳到 0x4 的地方执行程序
         *
         * 设置栈 sp 是指 und 的地址
```

```
                * sp_und 未设置, 先设置它

                *     /* 在 und 异常处理函数中有可能会修改 r0-r12,
所以先保存 */

                * lr 是异常处理完后的返回地址, 也要保存 */

                * 保存现场 */

                * 处理 und 异常 */

                * 恢复 sp

                * cpu 就会切换到之前的模式
                */
*/
.text
.global _start

_start:
   b reset  /* vector 0 : reset */
   b do_und /* vector 4 : und */

   und_addr:
          .word do_und

   do_und:
         /* 执行到这里之前:

          * 1.  lr_und 保存有被中断模式中的下一条即将执行的指
令的地址

          * 2.  SPSR_und 保存有被中断模式的 CPSR

          * 3.  CPSR 中的 M4-M0 被设置为 11011, 进入到 und 模
式

          * 4.  跳到 0x4 的地方执行程序
          */

         /* sp_und 未设置, 先设置它 */
```

```
            ldr sp, =0x34000000

            /* 在und异常处理函数中有可能会修改r0-r12, 所以先保
存 */
            /* lr是异常处理完后的返回地址, 也要保存 */
            stmdb sp!, {r0-r12, lr}

            /* 保存现场 */
            /* 处理und异常 */
            mrs r0, cpsr
            ldr r1, =und_string
            bl printException

            /* 恢复现场 */
            ldmia sp!, {r0-r12, pc}^  /* ^会把spsr的值恢复
到cpsr里 */

      und_string:
            .string "undefined instruction exception"
```

程序改进

源程序

```
.text
.global _start

_start:
   b reset  /* vector 0 : reset */
/*使用b命令跳转  相对跳转*/
   b do_und /* vector 4 : und */

do_und:
   /* 执行到这里之前:
```

```
  * 1. lr_und 保存有被中断模式中的下一条即将执行的指令的地址
  * 2. SPSR_und 保存有被中断模式的CPSR
  * 3. CPSR 中的M4-M0 被设置为11011, 进入到und 模式
  * 4. 跳到0x4 的地方执行程序
  */

/* sp_und 未设置, 先设置它 */
ldr sp, =0x34000000

/* 在und 异常处理函数中有可能会修改r0-r12, 所以先保存 */
/* lr 是异常处理完后的返回地址, 也要保存 */
stmdb sp!, {r0-r12, lr}

/* 保存现场 */
/* 处理und 异常 */
mrs r0, cpsr
ldr r1, =und_string

/*这里又使用bl 指令跳转, 如果是nand 启动, 这个函数在4k 之
外, 这个函数必定出错 为了保险, 跳转到sdram 中执行程序*/
bl printException

/* 恢复现场 */
ldmia sp!, {r0-r12, pc}^  /* ^会把spsr 的值恢复到cpsr
里 */

und_string:
      .string "undefined instruction exception"
```

```
reset:
    /* 关闭看门狗 */
    ldr r0, =0x53000000
    ldr r1, =0
    str r1, [r0]

    /* 设置MPLL, FCLK : HCLK : PCLK = 400m : 100m : 50m
*/
    /* LOCKTIME(0x4C000000) = 0xFFFFFFFF */
    ldr r0, =0x4C000000
    ldr r1, =0xFFFFFFFF
    str r1, [r0]

    /* CLKDIVN(0x4C000014) = 0X5, tFCLK:tHCLK:tPCLK =
1:4:8  */
    ldr r0, =0x4C000014
    ldr r1, =0x5
    str r1, [r0]

    /* 设置CPU工作于异步模式 */
    mrc p15,0,r0,c1,c0,0
    orr r0,r0,#0xc0000000   //R1_nF:OR:R1_iA
    mcr p15,0,r0,c1,c0,0

    /* 设置MPLLCON(0x4C000004) = (92<<12)|(1<<4)|(1<<0)
     *  m = MDIV+8 = 92+8=100
     *  p = PDIV+2 = 1+2 = 3
     *  s = SDIV = 1
     *  FCLK = 2*m*Fin/(p*2^s) = 2*100*12/(3*2^1)=400M
     */
    ldr r0, =0x4C000004
    ldr r1, =(92<<12)|(1<<4)|(1<<0)
    str r1, [r0]

    /* 一旦设置PLL, 就会锁定lock time直到PLL输出稳定
     * 然后CPU工作于新的频率FCLK
     */
```

```
	/* 设置内存: sp 栈 */
	/* 分辨是nor/nand 启动
	 * 写0 到0 地址, 再读出来
	 * 如果得到0, 表示0 地址上的内容被修改了, 它对应ram, 这就
是nand 启动
	 * 否则就是nor 启动
	 */
	mov r1, #0
	ldr r0, [r1] /* 读出原来的值备份 */
	str r1, [r1] /* 0->[0] */
	ldr r2, [r1] /* r2=[0] */
	cmp r1, r2   /* r1==r2? 如果相等表示是NAND 启动 */
	ldr sp, =0x40000000+4096 /* 先假设是nor 启动 */
	moveq sp, #4096  /* nand 启动 */
	streq r0, [r1]   /* 恢复原来的值 */

	bl sdram_init
	//bl sdram_init2    /* 用到有初始值的数组, 不是位置无关码
*/

	/* 重定位text, rodata, data 段整个程序 */
	bl copy2sdram

	/* 清除BSS 段 */
	bl clean_bss

	bl uart0_init
```

```
        bl print1
        /* 故意加入一条未定义指令 */
    und_code:
        .word 0xdeadc0de  /* 未定义指令 */
        bl print2

        //bl main  /* 使用BL命令相对跳转, 程序仍然在NOR/sram执
行 */

        ldr pc, =main  /* 绝对跳转, 跳到SDRAM */

    halt:
        b halt
```

改进后代码

```
    .text
    .global _start

    _start:
            b reset  /* vector 0 : reset */
    /*跳转到sdram执行这个函数，那么这个函数一定在sdram中
    我们需要指定让他去前面这块内存去读这个值，担心如果这个文件很大，
超过4Knand就没法去读这个文件*/
            ldr pc, und_addr /* vector 4 : und */

    /*增加如下 查看反汇编，在08的地址读让后跳到3c*/
    und_addr:
            .word do_und

    do_und:
            /* 执行到这里之前:
             * 1. lr_und保存有被中断模式中的下一条即将执行的指
令的地址
             * 2. SPSR_und保存有被中断模式的CPSR
```

```
                     * 3. CPSR 中的 M4-M0 被设置为 11011, 进入到 und 模
式
                     * 4. 跳到 0x4 的地方执行程序
                     */

                    /* sp_und 未设置, 先设置它 */
                    ldr sp, =0x34000000

                    /* 在 und 异常处理函数中有可能会修改 r0-r12, 所以先保
存 */
                    /* lr 是异常处理完后的返回地址, 也要保存 */
                    stmdb sp!, {r0-r12, lr}

                    /* 保存现场 */
                    /* 处理 und 异常 */
                    mrs r0, cpsr
                    ldr r1, =und_string
                    bl printException

                    /* 恢复现场 */
                    ldmia sp!, {r0-r12, pc}^  /* ^会把 spsr 的值恢复
到 cpsr 里 */

          und_string:
                    .string "undefined instruction exception"
          /**如果你的程序长度稍有变化，就不能保证运行
          加上 .align 4 才能保证后面的程序以 4 字节对齐，保证程序运行
          **/
          .align 4

          reset:
```

```
/* 关闭看门狗 */
ldr r0, =0x53000000
ldr r1, =0
str r1, [r0]

/* 设置MPLL, FCLK : HCLK : PCLK = 400m :
100m : 50m */
/* LOCKTIME(0x4C000000) = 0xFFFFFFFF */
ldr r0, =0x4C000000
ldr r1, =0xFFFFFFFF
str r1, [r0]

/* CLKDIVN(0x4C000014) = 0X5,
tFCLK:tHCLK:tPCLK = 1:4:8  */
ldr r0, =0x4C000014
ldr r1, =0x5
str r1, [r0]

/* 设置CPU工作于异步模式 */
mrc p15,0,r0,c1,c0,0
orr r0,r0,#0xc0000000   //R1_nF:OR:R1_iA
mcr p15,0,r0,c1,c0,0

/* 设置MPLLCON(0x4C000004) =
(92<<12)|(1<<4)|(1<<0)
 *  m = MDIV+8 = 92+8=100
 *  p = PDIV+2 = 1+2 = 3
 *  s = SDIV = 1
 *  FCLK = 2*m*Fin/(p*2^s) =
2*100*12/(3*2^1)=400M
 */
ldr r0, =0x4C000004
ldr r1, =(92<<12)|(1<<4)|(1<<0)
str r1, [r0]

/* 一旦设置PLL, 就会锁定lock time直到PLL输出稳
定
 * 然后CPU工作于新的频率FCLK
```

```
*/


/* 设置内存: sp 栈 */
/* 分辨是nor/nand 启动
 * 写0 到0 地址, 再读出来
 * 如果得到0, 表示0 地址上的内容被修改了, 它对应
ram, 这就是nand 启动
 * 否则就是nor 启动
 */
mov r1, #0
ldr r0, [r1] /* 读出原来的值备份 */
str r1, [r1] /* 0->[0] */
ldr r2, [r1] /* r2=[0] */
cmp r1, r2   /* r1==r2? 如果相等表示是NAND 启动
*/
ldr sp, =0x40000000+4096 /* 先假设是nor 启动 */
moveq sp, #4096  /* nand 启动 */
streq r0, [r1]   /* 恢复原来的值 */

bl sdram_init
//bl sdram_init2    /* 用到有初始值的数组, 不是位
置无关码 */

/* 重定位text, rodata, data 段整个程序 */
bl copy2sdram

/* 清除BSS 段 */
```

```
        bl clean_bss

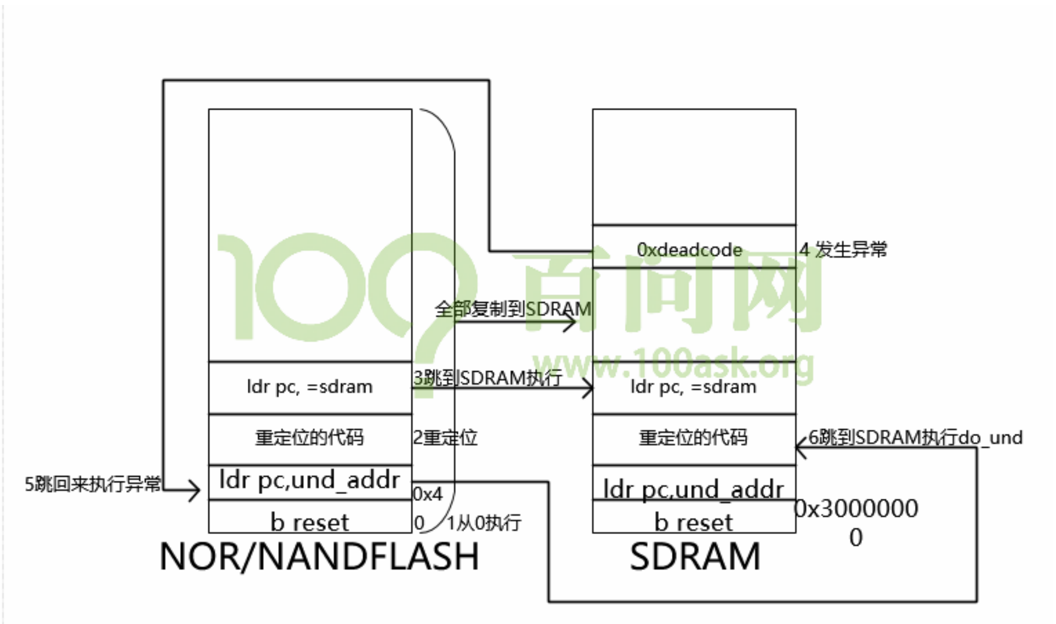
    /*把链接地址赋值给pc 直接就跳转到sdram 中*/
        ldr pc, =sdram
sdram:
        bl uart0_init

        bl print1
        /* 故意加入一条未定义指令 */
und_code:
        .word 0xdeadc0de  /* 未定义指令 */
        bl print2

        //bl main  /* 使用BL 命令相对跳转, 程序仍然在
NOR/sram 执行 */
        ldr pc, =main  /* 绝对跳转, 跳到SDRAM */

halt:
        b halt
```

看一下整个程序的执行过程



第 005 节_swi 异常模示程序示例

这节我们再来演示 swi 的处理流程 swi 软件中断:software interrupt 在前面的视频中我们讲过 ARMCPU 有 7 中模式，除了用户模式以外，其他 6 种都是特权模式，这些特权模式可以直接修改 CPSR 进入其他模式

usr 用户模式不能修改 CPSR 进入其他模式 Linux 应用程序一般运行于用户模式 APP 运行于 usermode，(受限模式，不可访问硬件)

APP 想访问硬件，必须切换模式，怎么切换 发生异常 3 种模式 中断是一种异常 und 也是 swi + 某个值(使用软中断切换模式)

现在 start.S 把要做的事情列出来

/*********1************************/ /* 复位之后, cpu 处于 svc 模式 * 现在, 切换到 usr 模式 * 设置栈 * 跳转执行 */

/********2 故意引入一条 swi 指令*********/

/******3 需在_start 这里放一条 swi 指令**********/

查看异常向量表 swi 异常的向量地址是 0x8

Table 2-3. Exception Vectors

Address	Exception	Mode in Entry
0x00000000	Reset	Supervisor
0x00000004	Undefined instruction	Undefined
0x00000008	Software Interrupt	Supervisor
0x0000000C	Abort (prefetch)	Abort
0x00000010	Abort (data)	Abort
0x00000014	Reserved	Reserved
0x00000018	IRQ	IRQ
0x0000001C	FIQ	FIQ

我们先切换到 usr 模式下

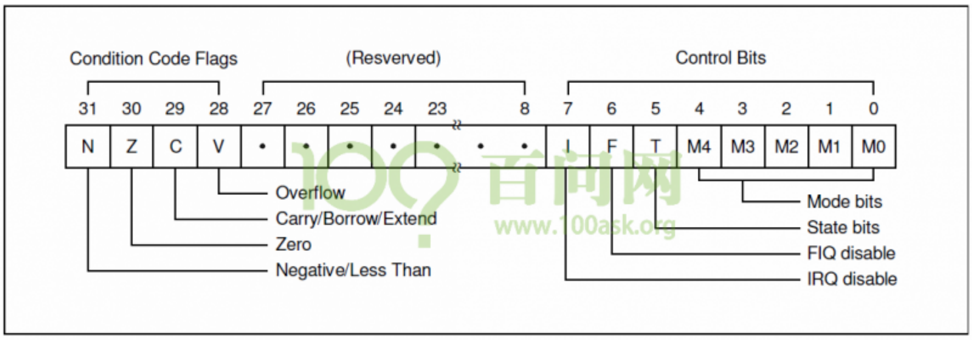


Figure 2-6. Program Status Register Format

Table 2-1. PSR Mode Bit Values

M[4:0]	Mode	Visible THUMB state registers	Visible ARM state registers
10000	User	R7..R0, LR, SP PC, CPSR	R14..R0, PC, CPSR
10001	FIQ	R7..R0, LR_fiq, SP_fiq PC, CPSR, SPSR_fiq	R7..R0, R14_fiq..R8_fiq, PC, CPSR, SPSR_fiq
10010	IRQ	R7..R0, LR_irq, SP_irq PC, CPSR, SPSR_irq	R12..R0, R14_irq, R13_irq, PC, CPSR, SPSR_irq
10011	Supervisor	R7..R0, LR_svc, SP_svc, PC, CPSR, SPSR_svc	R12..R0, R14_svc, R13_svc, PC, CPSR, SPSR_svc
10111	Abort	R7..R0, LR_abt, SP_abt, PC, CPSR, SPSR_abt	R12..R0, R14_abt, R13_abt, PC, CPSR, SPSR_abt
11011	Undefined	R7..R0 LR_und, SP_und, PC, CPSR, SPSR_und	R12..R0, R14_und, R13_und, PC, CPSR
11111	System	R7..R0, LR, SP PC, CPSR	R14..R0, PC, CPSR

usr 模式下的 M0 ~ M4 是 10000

/********5 先进入 usr 模式*/ mrs r0, cpsr /* 读出 cpsr 读到 r0 */ / 使用 bic 命令 bitclean 把低 4 位清零/ bic r0, r0, #0xf /* 修改 M4-M0 为 0b10000, 进入 usr 模式 */ msr cpsr, r0

/************6 设置栈*/ /* 设置 sp_usr */ ldr sp, =0x33f00000

编译运行 发现可以处理 und 指令

添加 swi 异常，仿照未定义指令做

```
.text
.global _start

_start:
   b reset          /* vector 0 : reset */
   ldr pc, und_addr /* vector 4 : und */
/*****************1

添加 swi 指令

*/
```

```
    ldr pc, swi_addr /* vector 8 : swi */

und_addr:
    .word do_und

/***************2


仿照 und 未定义添加指令

*/
swi_addr:
    .word do_swi

do_und:
    /* 执行到这里之前:

     * 1. lr_und 保存有被中断模式中的下一条即将执行的指令的地址

     * 2. SPSR_und 保存有被中断模式的 CPSR

     * 3. CPSR 中的 M4-M0 被设置为 11011, 进入到 und 模式

     * 4. 跳到 0x4 的地方执行程序
     */

    /* sp_und 未设置, 先设置它 */
    ldr sp, =0x34000000

    /* 在 und 异常处理函数中有可能会修改 r0-r12, 所以先保存 */

    /* lr 是异常处理完后的返回地址, 也要保存 */
    stmdb sp!, {r0-r12, lr}

    /* 保存现场 */

    /* 处理 und 异常 */
    mrs r0, cpsr
    ldr r1, =und_string
    bl printException
```

```
        /* 恢复现场 */

        ldmia sp!, {r0-r12, pc}^  /* ^会把spsr的值恢复到cpsr
里 */

    und_string:
        .string "undefined instruction exception"

    /*************3

    复制do_und

    修改为swi

    */
    do_swi:
        /* 执行到这里之前:

         * 3.1. lr_svc保存有被中断模式中的下一条即将执行的指令的地
址

         * 3.2. SPSR_svc保存有被中断模式的CPSR

         * 3.3. CPSR中的M4-M0被设置为10011, 进入到svc模式

         * 3.4. 跳到0x08的地方执行程序

         */

        /* 3.5 sp_svc未设置, 先设置它 */

        ldr sp, =0x33e00000

        /* 3.6 在swi异常处理函数中有可能会修改r0-r12, 所以先保存
*/

        /* 3.7 lr是异常处理完后的返回地址, 也要保存 */

        stmdb sp!, {r0-r12, lr}

        /* 3.8 保存现场 */
```

```
    /* 3.9 处理swi 异常 只是打印 */
    mrs r0, cpsr
    ldr r1, =swi_string
    bl printException

    /*3.10  恢复现场 */

    ldmia sp!, {r0-r12, pc}^  /* ^会把spsr 的值恢复到cpsr
里 */
/*****

swi 处理函数

*/
swi_string:
    .string "swi exception"
```

上传代码实验 烧写 发现没有执行 我们先把下面这些代码注释掉

```
/*************3

复制do_und

修改为swi

*/

    /* 执行到这里之前:

     * 3.1. lr_svc 保存有被中断模式中的下一条即将执行的指令的地
址

     * 3.2. SPSR_svc 保存有被中断模式的CPSR

     * 3.3. CPSR 中的M4-M0 被设置为10011, 进入到svc 模式

     * 3.4. 跳到0x08 的地方执行程序
     */

    /* 3.5 sp_svc 未设置, 先设置它 */
    ldr sp, =0x33e00000
```

```
        /* 3.6 在swi 异常处理函数中有可能会修改r0-r12, 所以先保存
*/
        /* 3.7 lr 是异常处理完后的返回地址, 也要保存 */
        stmdb sp!, {r0-r12, lr}

        /* 3.8 保存现场 */

        /* 3.9 处理swi 异常 只是打印 */
        mrs r0, cpsr
        ldr r1, =swi_string
        bl printException

        /*3.10  恢复现场 */

        ldmia sp!, {r0-r12, pc}^  /* ^会把spsr 的值恢复到cpsr
里 */
    /*****
    swi 处理函数
    */
    swi_string:
        .string "swi exception"
```

上传编译 烧写执行 可以正常运行

循环打印 swi 0x123 /* 执行此命令, 触发 SWI 异常, 进入 0x8 执行 */ 执行后继续执行 ldr pc, swi_addr /* vector 8 : swi */

表明问题出现在 do_swi:函数中 先把下面这句话注释掉 .string "swi exception"

编译烧写运行 程序可以正常运行

显然程序问题出现在 .string "swi exception" 这句话，为什么加上这句话程序就无法执行，查看一下反汇编

```
    30000064 <swi_string>: //这里地址是64

    30000064: 20697773      rsbcs  r7, r9, r3, ror r7
    30000068: 65637865      strvsb r7, [r3, #-2149]!
    3000007c: 6f697470      swivs  0x00697470
    30000070: 0000006e      andeq  r0, r0, lr, rrx
```

```
30000082 <reset>: //我们使用的是ARM 指令集，应该是4 字节对齐，发现这里并不是，问题就在这里
30000082: e3a00453     mov    r0, #1392508928     ; 0x53000000
30000086: e3a01000     mov    r1, #0 ; 0x0
3000008a: e5801000     str    r1, [r0]
3000008e: e3a00313     mov    r0, #1275068416     ; 0x4c000000
30000092: e3e01000     mvn    r1, #0 ; 0x0
```

因为这个字符串长度有问题
前面 und_string 那里的字符串长度刚刚好
我们不能把问题放在运气上面 添加
/******* 以 4 字节对齐

- /

```
.align 4

do_swi:
    /* 执行到这里之前:
     * 3.1. lr_svc 保存有被中断模式中的下一条即将执行的指令的地址
     * 3.2. SPSR_svc 保存有被中断模式的CPSR
     * 3.3. CPSR 中的M4-M0 被设置为10011, 进入到svc 模式
     * 3.4. 跳到0x08 的地方执行程序
     */

    /* 3.5 sp_svc 未设置, 先设置它 */
    ldr sp, =0x33e00000

    /* 3.6 在swi 异常处理函数中有可能会修改r0-r12, 所以先保存 */
    /* 3.7 lr 是异常处理完后的返回地址, 也要保存 */
    stmdb sp!, {r0-r12, lr}
```

```
    /* 3.8 保存现场 */

    /* 3.9 处理swi 异常 只是打印 */
    mrs r0, cpsr
    ldr r1, =swi_string
    bl printException

    /*3.10  恢复现场 */

    ldmia sp!, {r0-r12, pc}^  /* ^会把spsr 的值恢复到cpsr
里 */
  /*****
  swi 处理函数
  */
  swi_string:
    .string "swi exception"


  .align 4
  /**************
  表明下面的标号要放在4 字节对齐的地方
  */
```

上传代码编译运行查看反汇编

```
30000068 <swi_string>:
30000068: 20697773    rsbcs r7, r9, r3, ror r7
3000006c: 65637865    strvsb r7, [r3, #-2149]!
30000070: 6f697470    swivs 0x00697470
30000074: 0000006e    andeq r0, r0, lr, rrx
    ...

30000080 <reset>: //现在reset 放在4 自己对齐的地方

30000080: e3a00453    mov   r0, #1392508928    ;
0x53000000
30000084: e3a01000    mov   r1, #0 ; 0x0
30000088: e5801000    str   r1, [r0]
```

```
    3000008c: e3a00313      mov     r0, #1275068416     ;
0x4c000000
    30000090: e3e01000      mvn     r1, #0 ; 0x0
    30000094: e5801000      str     r1, [r0]
    30000098: e59f0084      ldr     r0, [pc, #132]      ;
30000124 <.text+0x124>
    3000009c: e3a01005      mov     r1, #5 ; 0x5
    300000a0: e5801000      str     r1, [r0]
    300000a4: ee110f10      mrc     15, 0, r0, cr1, cr0, {0}
    300000a8: e3800103      orr     r0, r0, #-1073741824
        ; 0xc0000000
    300000ac: ee010f10      mcr     15, 0, r0, cr1, cr0, {0}
    300000b0: e59f0070      ldr     r0, [pc, #112]      ;
30000128 <.text+0x128>
    300000b4: e59f1070      ldr     r1, [pc, #112]      ;
3000012c <.text+0x12c>
    300000b8: e5801000      str     r1, [r0]
    300000bc: e3a01000      mov     r1, #0 ; 0x0
    300000c0: e5910000      ldr     r0, [r1]
```

下载烧写 程序执行完全没有问题 程序备份修改代码

swi 可以根据应用程序传入的 val 来判断为什么调用 swi 指令，我们的异常处理函数能不能把这个 val 值读出来

```
do_swi:
    /* 执行到这里之前:

     * 1. lr_svc 保存有被中断模式中的下一条即将执行的指令的地址

     * 2. SPSR_svc 保存有被中断模式的 CPSR

     * 3. CPSR 中的 M4-M0 被设置为 10011, 进入到 svc 模式

     * 4. 跳到 0x08 的地方执行程序
     */


    /* sp_svc 未设置, 先设置它 */
    ldr sp, =0x33e00000


    /* 保存现场 */

    /* 在 swi 异常处理函数中有可能会修改 r0-r12, 所以先保存 */
```

```
    /* lr 是异常处理完后的返回地址, 也要保存 */
    stmdb sp!, {r0-r12, lr}

/********2
我们得把 lr 拿出来用

我们要把 lr 拿出来保存

因为bl printException 会破坏lr

mov rX, lr

我把 lr 保存在那个寄存器?

这个函数 bl printException 可能会修改某些寄存器,但是又会恢复这些寄存器,我得知道他会保护那些寄存器

我们之前讲过ATPCS 规则

对于 r4 ~ r11 在C 函数里他都会保存这几个寄存器,如果用到的话就把他保存起来,执行完C 函数再把它释放掉

我们把 lr 保存在r4 寄存器里,r4 寄存器不会被C 语言破坏
*/
    mov r4, lr

    /* 处理swi 异常 */
    mrs r0, cpsr
    ldr r1, =swi_string
    bl printException

/********1
跳转到printSWIVal

如何才能知道swi 的值呢?
```

```
    我们得读出swi 0x123指令，这条指令保存在内存中，我们得找到他的内存地址
    执行完0x123指令以后，会发生一次异常，那个异常模式里的lr寄存器会保存下一条指令的地址
    我们把lr寄存器的地址减去4就是swi 0x123这条指令的地址

    */
    /**********3
    我再把r4的寄存器赋给r0让后打印
    我们得写出打印函数
    mov r0, r4
    指令地址减4才可以
    swi 0x123
    下一条指令bl main 减4就是指令本身
    */
       sub r0, r4, #4
       bl printSWIVal

       /* 恢复现场 */

       ldmia sp!, {r0-r12, pc}^  /* ^会把spsr的值恢复到cpsr里 */

    swi_string:
       .string "swi exception"




    <syntaxhighlight lang="c" >
    在uart.c添加printSWIVal打印函数

    void printSWIVal(unsigned int *pSWI)
    {
```

```
    puts("SWI val = ");
    printHEx(*pSWI & ~0xff000000); //高 8 位忽略掉
    puts("\n\r");
}
```

编译实验运行没有问题



我们再来看看这个程序是怎么跳转的

/**************1 发生 swi 异常，他是在 sdram 中，CPU 就会跳到 0x8 的地方 swi 0x123 /* 执行此命令，触发 SWI 异常，进入 0x8 执行 */

- /

/************2
_start: b reset /* vector 0 : reset */ ldr pc, und_addr /* vector 4 : und */ 执行这条读内存指令 ldr pc, swi_addr /* vector 8 : swi */
读到 swi_addr 地址跳转到 sdram 执行代码 do_swi 那段代码
swi_addr: .word do_swi

- /

/************3 这段代码被设置栈保存现场 调用处理函数恢复现场，让后就会跳到 sdram 执行 swi 0x123 的下一条指令

```
do_swi:
    /* 执行到这里之前:
     * 1. lr_svc 保存有被中断模式中的下一条即将执行的指令的地址
     * 2. SPSR_svc 保存有被中断模式的 CPSR
     * 3. CPSR 中的 M4-M0 被设置为 10011, 进入到 svc 模式
     * 4. 跳到 0x08 的地方执行程序
     */
```

```
/* sp_svc 未设置, 先设置它 */
ldr sp, =0x33e00000

/* 保存现场 */
/* 在 swi 异常处理函数中有可能会修改 r0-r12, 所以先保存 */
/* lr 是异常处理完后的返回地址, 也要保存 */
stmdb sp!, {r0-r12, lr}

mov r4, lr

/* 处理 swi 异常 */
mrs r0, cpsr
ldr r1, =swi_string
bl printException

sub r0, r4, #4
bl printSWIVal

/* 恢复现场 */
ldmia sp!, {r0-r12, pc}^  /* ^会把 spsr 的值恢复到 cpsr
里 */

swi_string:
	.string "swi exception"
*/
```

这节视频我们讲解了 swi 的处理流程

第 006 节_按键中断程序示例_概述与初始

在前面的视频里我们举了一个例子，母亲看书被声音打断，远处的声音来源有多种多样，声音传入耳朵，再由耳朵传入大脑，整个过程涉及声音来源耳朵大脑，为了确保这个母亲看书的过程能够被声音打断， 我们必须保证声音来源可以发出声音，耳朵没有聋，脑袋没有傻

- 类比嵌入式系统我们可以设置中断源，让他发出中断信号，还需要设置中断控制器，让他把这些信号发送给 CPU，还需要设置 CPU 让他能够处理中断

中断的处理流程

1 中断初始化： 1 我们需要设置中断源，让它能够发出中断喜好 2 设置中断控制器，让它能发出中断给 CPU 3 设置 CPU，CPSR 有 I 位，是总开关 我们需要这样设置，中断源才能发送给 CPU

2 处理完要清中断

3 处理时，要分辨中断源，对于不同的中断源要执行不同的处理函数

下面开始写代码

打开 start.S 先做初始化工作，先做第 3 设置 CPU，CPSR 有 I 位，是总开关 我们需要把 CPSR 寄存器 bit7 给清零，这是中断的总开关，如果 bit7 设置为 1 CPU 无法响应任何中
断

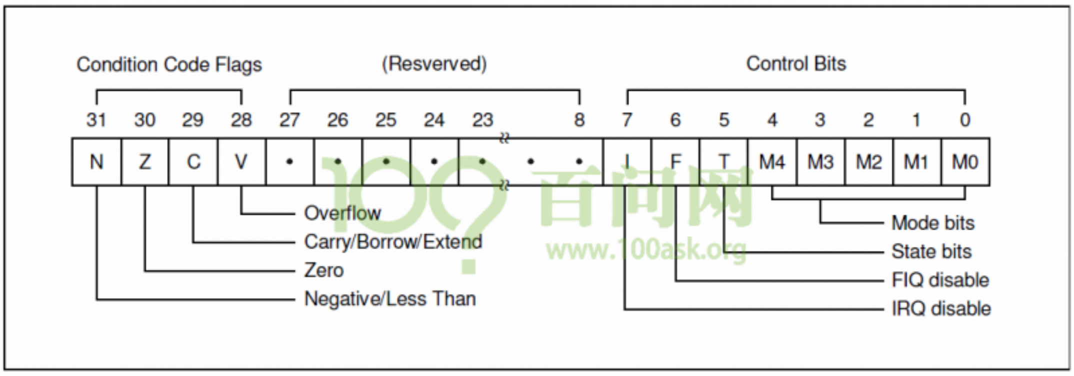


Figure 2-6. Program Status Register Format

```
/* 清除BSS段 */
bl clean_bss

/* 复位之后, cpu处于svc模式
 * 现在, 切换到usr模式
 */
mrs r0, cpsr         /* 读出cpsr */
bic r0, r0, #0xf     /* 修改M4-M0为0b10000, 进入usr
模式 */
/****************1
把bit7这一位清零
```

```
*/
    bic r0, r0, #(1<<7)  /* 清除I位, 使能中断 */
    msr cpsr, r0

    /* 设置 sp_usr */
    ldr sp, =0x33f00000

    ldr pc, =sdram
sdram:
    bl uart0_init

    bl print1
    /* 故意加入一条未定义指令 */
und_code:
    .word 0xdeadc0de  /* 未定义指令 */
    bl print2

    swi 0x123  /* 执行此命令, 触发SWI异常, 进入0x8执行 */
/**************2
调用两个中断
*/
    bl interrupt_init /*初始化中断控制器*/

    bl eint_init /*初始化按键, 设为中断源*/

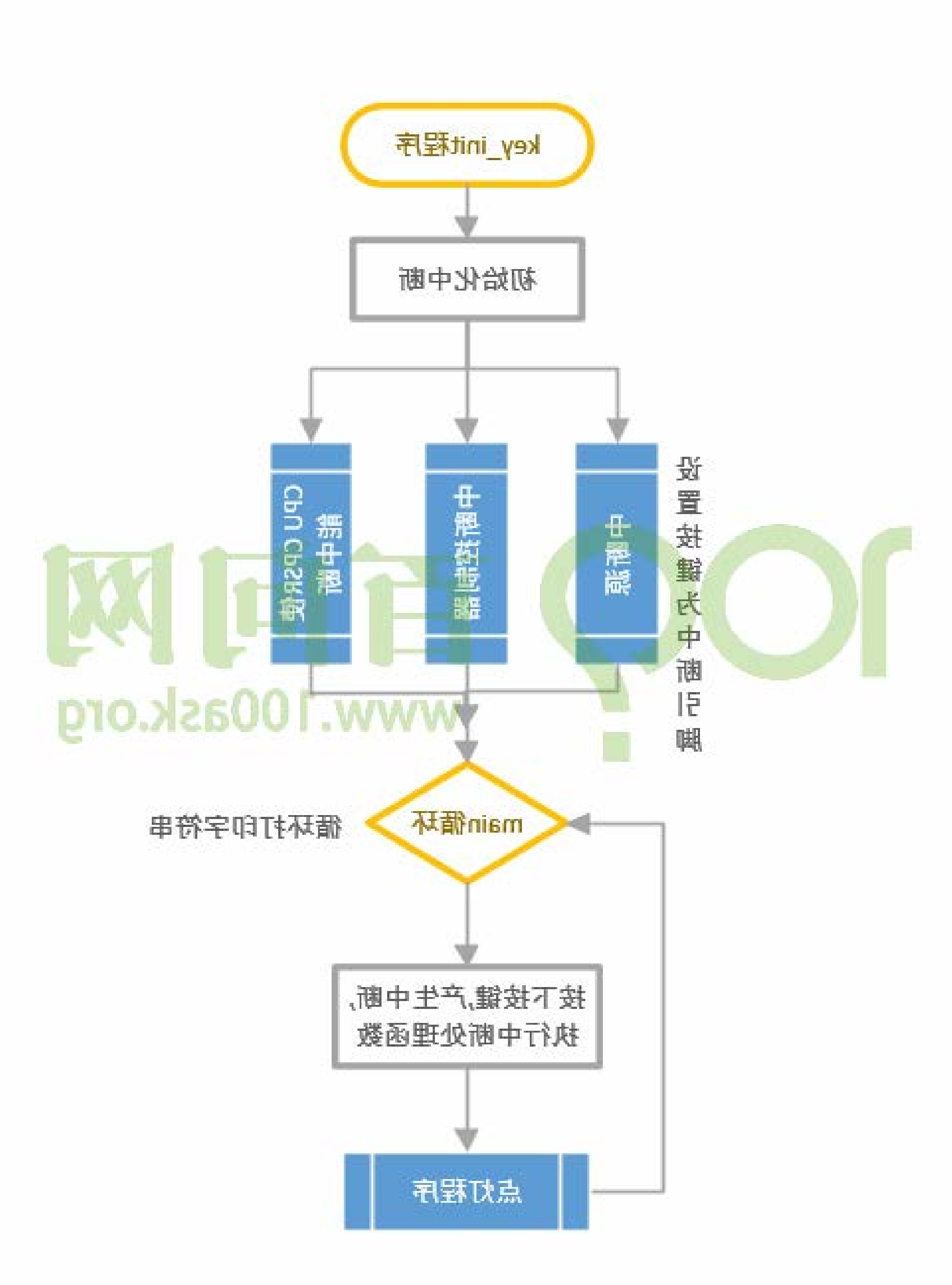
/*******需要初始化上面这两个函数*/

    //bl main  /* 使用BL命令相对跳转, 程序仍然在NOR/sram执
行 */

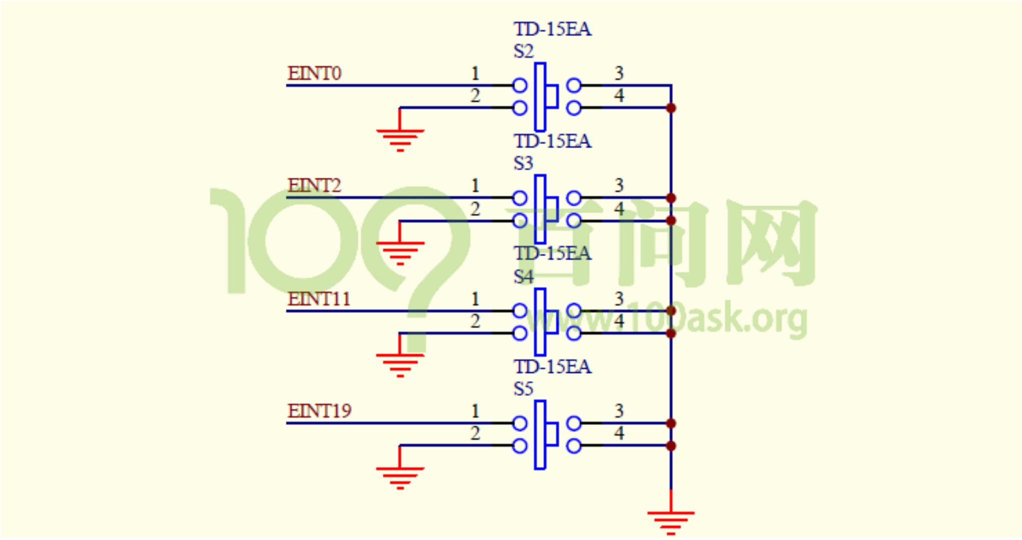
    ldr pc, =main  /* 绝对跳转, 跳到SDRAM */

halt:
    b halt
```

添加一个 [illegible] 文件 这个程序稍微有些复杂，我们先画个流程图



我们想达到按下按键灯亮松开按键灯灭 把下面四个按键全部配置为外部中断按键



打开芯片手册找到第九章 IO ports 直接搜索 EINT0 号中断和 EINT2 号中断 找配置寄存器 GPFCON

Register	Address	R/W	Description	Reset Value
GPFCON	0x56000050	R/W	Configures the pins of port F	0x0
GPFDAT	0x56000054	R/W	The data register for port F	Undef.
GPFUP	0x56000058	R/W	Pull-up disable register for port F	0x000
Reserved	0x5600005c	–	–	–

GPFCON	Bit	Description
GPF7	[15:14]	00 = Input 01 = Output 10 = EINT[7] 11 = Reserved
GPF6	[13:12]	00 = Input 01 = Output 10 = EINT[6] 11 = Reserved
GPF5	[11:10]	00 = Input 01 = Output 10 = EINT[5] 11 = Reserved
GPF4	[9:8]	00 = Input 01 = Output 10 = EINT[4] 11 = Reserved
GPF3	[7:6]	00 = Input 01 = Output 10 = EINT[3] 11 = Reserved
GPF2	[5:4]	00 = Input 01 = Output 10 = EINT2] 11 = Reserved
GPF1	[3:2]	00 = Input 01 = Output 10 = EINT[1] 11 = Reserved
GPF0	[1:0]	00 = Input 01 = Output 10 = EINT[0] 11 = Reserved

为了简单操作

```
/* 初始化按键, 设为中断源 */
void key_eint_init(void)
{
/*1 配置 GPIO 为中断引脚 */
//先把 eint0 和 eint2 这两个引脚清零
GPFCON &= ~((3<<0) | (3<<4));
GPFCON |= ((2<<0) | (2<<4));    /* S2,S3 被配置为中断引脚 */
```

通过电路图得知，S4、S5 按键为 EINT11 号中断引脚和 EINT19 号中断引脚

Register	Address	R/W	Description	Reset Value
GPGCON	0x56000060	R/W	Configures the pins of port G	0x0
GPGDAT	0x56000064	R/W	The data register for port G	Undef.
GPGUP	0x56000068	R/W	Pull-up disable register for port G	0xfc00

GPGCON	Bit	Description	
GPG15*	[31:30]	00 = Input 10 = EINT[23]	01 = Output 11 = Reserved
GPG14*	[29:28]	00 = Input 10 = EINT[22]	01 = Output 11 = Reserved
GPG13*	[27:26]	00 = Input 10 = EINT[21]	01 = Output 11 = Reserved
GPG12	[25:24]	00 = Input 10 = EINT[20]	01 = Output 11 = Reserved
GPG11	[23:22]	00 = Input 10 = EINT[19]	01 = Output 11 = TCLK[1]
GPG10	[21:20]	00 = Input 10 = EINT[18]	01 = Output 11 = nCTS1
GPG9	[19:18]	00 = Input 10 = EINT[17]	01 = Output 11 = nRTS1
GPG8	[17:16]	00 = Input 10 = EINT[16]	01 = Output 11 = Reserved
GPG7	[15:14]	00 = Input 10 = EINT[15]	01 = Output 11 = SPICLK1
GPG6	[13:12]	00 = Input 10 = EINT[14]	01 = Output 11 = SPIMOSI1
GPG5	[11:10]	00 = Input 10 = EINT[13]	01 = Output 11 = SPIMISO1
GPG4	[9:8]	00 = Input 10 = EINT[12]	01 = Output 11 = LCD_PWRDN
GPG3	[7:6]	00 = Input 10 = EINT[11]	01 = Output 11 = nSS1
GPG2	[5:4]	00 = Input 10 = EINT[10]	01 = Output 11 = nSS0
GPG1	[3:2]	00 = Input 10 = EINT[9]	01 = Output 11 = Reserved
GPG0	[1:0]	00 = Input 10 = EINT[8]	01 = Output 11 = Reserved

```
GPGCON &= ~((3<<6) | (3<<11));
GPGCON |= ((2<<6) | (2<<11));    /* S4,S5 被配置为中断引脚 */
```

/*2 设置中断触发方式：(按下松开，从低电源变为高电源，或者从)双边沿触发 */

Register	Address	R/W	Description	Reset Value
EXTINT0	0x56000088	R/W	External interrupt control register 0	0x000000
EXTINT1	0x5600008c	R/W	External interrupt control register 1	0x000000
EXTINT2	0x56000090	R/W	External interrupt control register 2	0x000000

EXTINT0	Bit	Description
EINT7	[30:28]	Setting the signaling method of the EINT7. 000 = Low level 001 = High level 01x = Falling edge triggered 10x = Rising edge triggered 11x = Both edge triggered
EINT6	[26:24]	Setting the signaling method of the EINT6. 000 = Low level 001 = High level 01x = Falling edge triggered 10x = Rising edge triggered 11x = Both edge triggered
EINT5	[22:20]	Setting the signaling method of the EINT5. 000 = Low level 001 = High level 01x = Falling edge triggered 10x = Rising edge triggered 11x = Both edge triggered
EINT4	[18:16]	Setting the signaling method of the EINT4. 000 = Low level 001 = High level 01x = Falling edge triggered 10x = Rising edge triggered 11x = Both edge triggered
EINT3	[14:12]	Setting the signaling method of the EINT3. 000 = Low level 001 = High level 01x = Falling edge triggered 10x = Rising edge triggered 11x = Both edge triggered
EINT2	[10:8]	Setting the signaling method of the EINT2. 000 = Low level 001 = High level 01x = Falling edge triggered 10x = Rising edge triggered 11x = Both edge triggered
EINT1	[6:4]	Setting the signaling method of the EINT1. 000 = Low level 001 = High level 01x = Falling edge triggered 10x = Rising edge triggered 11x = Both edge triggered
EINT0	[2:0]	Setting the signaling method of the EINT0. 000 = Low level 001 = High level 01x = Falling edge triggered 10x = Rising edge triggered 11x = Both edge triggered

//设置 EINT0 EINT2 为双边沿触发

EXTINT0 |= (7<<0) | (7<<8); /* S2,S3 */

//设置 EINT11 为双边沿触发

EXTINT1	Bit	Description
FLTEN15	[31]	Filter enable for EINT15 0 = Filter Disable 1 = Filter Enable
EINT15	[30:28]	Setting the signaling method of the EINT15. 000 = Low level 001 = High level 01x = Falling edge triggered 10x = Rising edge triggered 11x = Both edge triggered
FLTEN14	[27]	Filter enable for EINT14 0 = Filter Disable 1 = Filter Enable
EINT14	[26:24]	Setting the signaling method of the EINT14. 000 = Low level 001 = High level 01x = Falling edge triggered 10x = Rising edge triggered 11x = Both edge triggered
FLTEN13	[23]	Filter enable for EINT13 0 = Filter Disable 1 = Filter Enable
EINT13	[22:20]	Setting the signaling method of the EINT13. 000 = Low level 001 = High level 01x = Falling edge triggered 10x = Rising edge triggered 11x = Both edge triggered
FLTEN12	[19]	Filter enable for EINT12 0 = Filter Disable 1 = Filter Enable
EINT12	[18:16]	Setting the signaling method of the EINT12. 000 = Low level 001 = High level 01x = Falling edge triggered 10x = Rising edge triggered 11x = Both edge triggered
FLTEN11	[15]	Filter enable for EINT11 0 = Filter Disable 1 = Filter Enable
EINT11	[14:12]	Setting the signaling method of the EINT11. 000 = Low level 001 = High level 01x = Falling edge triggered 10x = Rising edge triggered 11x = Both edge triggered
FLTEN10	[11]	Filter enable for EINT10 0 = Filter Disable 1 = Filter Enable
EINT10	[10:8]	Setting the signaling method of the EINT10. 000 = Low level 001 = High level 01x = Falling edge triggered 10x = Rising edge triggered 11x = Both edge triggered
FLTEN9	[7]	Filter enable for EINT9 0 = Filter Disable 1 = Filter Enable
EINT9	[6:4]	Setting the signaling method of the EINT9. 000 = Low level 001 = High level 01x = Falling edge triggered 10x = Rising edge triggered 11x = Both edge triggered
FLTEN8	[3]	Filter enable for EINT8 0 = Filter Disable 1 = Filter Enable
EINT8	[2:0]	Setting the signaling method of the EINT8. 000 = Low level 001 = High level 01x = Falling edge triggered 10x = Rising edge triggered 11x = Both edge triggered


```
EXTINT1 |= (7<<12);                /* S4 */
```

//设置 EINT19 为双边沿触发

EXTINT2	Bit	Description	Reset Value
FLTEN23	[31]	Filter enable for EINT23 0 = Filter Disable 1= Filter Enable	0
EINT23	[30:28]	Setting the signaling method of the EINT23. 000 = Low level 001 = High level 01x = Falling edge triggered 10x = Rising edge triggered 11x = Both edge triggered	000
FLTEN22	[27]	Filter Enable for EINT22 0 = Filter Disable 1= Filter Enable	0
EINT22	[26:24]	Setting the signaling method of the EINT22. 000 = Low level 001 = High level 01x = Falling edge triggered 10x = Rising edge triggered 11x = Both edge triggered	000
FLTEN21	[23]	Filter Enable for EINT21 0 = Filter Disable 1= Filter Enable	0
EINT21	[22:20]	Setting the signaling method of the EINT21. 000 = Low level 001 = High level 01x = Falling edge triggered 10x = Rising edge triggered 11x = Both edge triggered	000
FLTEN20	[19]	Filter Enable for EINT20 0 = Filter Disable 1= Filter Enable	0
EINT20	[18:16]	Setting the signaling method of the EINT20. 000 = Low level 001 = High level 01x = Falling edge triggered 10x = Rising edge triggered 11x = Both edge triggered	000
FLTEN19	[15]	Filter enable for EINT19 0 = Filter Disable 1= Filter Enable	0
EINT19	[14:12]	Setting the signaling method of the EINT19. 000 = Low level 001 = High level 01x = Falling edge triggered 10x = Rising edge triggered 11x = Both edge triggered	000
FLTEN18	[11]	Filter enable for EINT18 0 = Filter Disable 1= Filter Enable	0
EINT18	[10:8]	Setting the signaling method of the EINT18. 000 = Low level 001 = High level 01x = Falling edge triggered 10x = Rising edge triggered 11x = Both edge triggered	000

EXTINT2 |= (7<<12);   /* S5 */

/*外部中断屏蔽寄存器 EINTMASK，设置为 1 的话就禁止向外部发出中断信号，只有 EINTMASK 相应的位设置为 0 外部中断才能给中断控制器发信号 我们

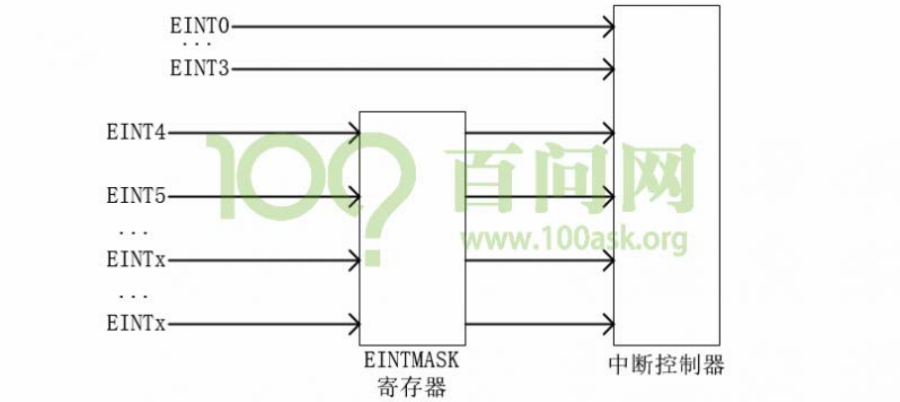
需要设置这个寄存器

EINTMASK (External Interrupt Mask Register)

Register	Address	R/W	Description	Reset Value
EINTMASK	0x560000a4	R/W	External interrupt mask register	0x000fffff

EINTMASK	Bit	Description
EINT23	[23]	0 = enable interrupt 1= masked
EINT22	[22]	0 = enable interrupt 1= masked
EINT21	[21]	0 = enable interrupt 1= masked
EINT20	[20]	0 = enable interrupt 1= masked
EINT19	[19]	0 = enable interrupt 1= masked
EINT18	[18]	0 = enable interrupt 1= masked
EINT17	[17]	0 = enable interrupt 1= masked
EINT16	[16]	0 = enable interrupt 1= masked
EINT15	[15]	0 = enable interrupt 1= masked
EINT14	[14]	0 = enable interrupt 1= masked
EINT13	[13]	0 = enable interrupt 1= masked
EINT12	[12]	0 = enable interrupt 1= masked
EINT11	[11]	0 = enable interrupt 1= masked
EINT10	[10]	0 = enable interrupt 1= masked
EINT9	[9]	0 = enable interrupt 1= masked
EINT8	[8]	0 = enable interrupt 1= masked
EINT7	[7]	0 = enable interrupt 1= masked
EINT6	[6]	0 = enable interrupt 1= masked
EINT5	[5]	0 = enable interrupt 1= masked
EINT4	[4]	0 = enable interrupt 1= masked
Reserved	[3:0]	Reserved

把EINT11设置为0 把EINT19设置为0对于EINT0 和EINT2显示为保留，默认时使能的，可以直接发送给中断控制器，无需设



置

/* 设置EINTMASK使能eint11,19 */ EINTMASK &= ~((1<<11) | (1<<19)); }

/* 读EINTPEND分辨率哪个EINT产生(eint4~23)(并且要清除他)

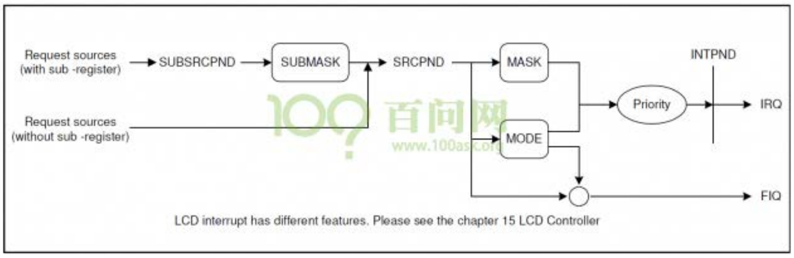
* 清除中断时，写EINTPEND的相应位

EINTPEND (External Interrupt Pending Register)

Register	Address	R/W	Description	Reset Value
EINTPEND	0x560000a8	R/W	External interrupt pending register	0x00

EINTPEND	Bit	Description	Reset Value
EINT23	[23]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
EINT22	[22]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
EINT21	[21]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
EINT20	[20]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
EINT19	[19]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
EINT18	[18]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
EINT17	[17]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
EINT16	[16]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
EINT15	[15]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
EINT14	[14]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
EINT13	[13]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
EINT12	[12]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
EINT11	[11]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
EINT10	[10]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
EINT9	[9]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
EINT8	[8]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
EINT7	[7]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
EINT6	[6]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
EINT5	[5]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
EINT4	[4]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
Reserved	[3:0]	Reserved	0000

我们接下来需要阅读第 14 章 Interrupt Controller 章节设置中断控制器我们只需要按照下面这张流程图设置就可以



了

我们需要设置 MASK 屏蔽寄存器 INTPND 等待处理，我们可以读这个寄存器，确定是那个中断产生了 SRCPND 不同的中断类型不可以直接到达这里执行

我们来看一下外部中断属于哪一种 打开芯片手册，从上往下读

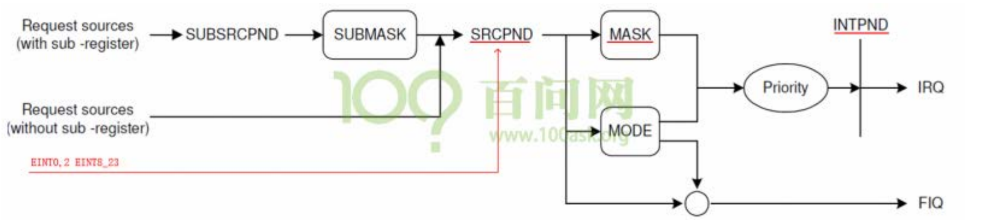
INTERRUPT SOURCES

The interrupt controller supports 60 interrupt sources as shown in the table below.

Sources	Descriptions	Arbiter Group
INT_ADC	ADC EOC and Touch interrupt (INT_ADC_S/INT_TC)	ARB5
INT_RTC	RTC alarm interrupt	ARB5
INT_SPI1	SPI1 interrupt	ARB5
INT_UART0	UART0 Interrupt (ERR, RXD, and TXD)	ARB5
INT_IIC	IIC interrupt	ARB4
INT_USBH	USB Host interrupt	ARB4
INT_USBD	USB Device interrupt	ARB4
INT_NFCON	Nand Flash Control Interrupt	ARB4
INT_UART1	UART1 Interrupt (ERR, RXD, and TXD)	ARB4
INT_SPI0	SPI0 interrupt	ARB4
INT_SDI	SDI interrupt	ARB 3
INT_DMA3	DMA channel 3 interrupt	ARB3
INT_DMA2	DMA channel 2 interrupt	ARB3
INT_DMA1	DMA channel 1 interrupt	ARB3
INT_DMA0	DMA channel 0 interrupt	ARB3
INT_LCD	LCD interrupt (INT_FrSyn and INT_FiCnt)	ARB3
INT_UART2	UART2 Interrupt (ERR, RXD, and TXD)	ARB2
INT_TIMER4	Timer4 interrupt	ARB2
INT_TIMER3	Timer3 interrupt	ARB2
INT_TIMER2	Timer2 interrupt	ARB2
INT_TIMER1	Timer1 interrupt	ARB 2
INT_TIMER0	Timer0 interrupt	ARB2
INT_WDT_AC97	Watch-Dog timer interrupt(INT_WDT, INT_AC97)	ARB1
INT_TICK	RTC Time tick interrupt	ARB1
nBATT_FLT	Battery Fault interrupt	ARB1
INT_CAM	Camera Interface (INT_CAM_C, INT_CAM_P)	ARB1
EINT8_23	External interrupt 8 – 23	ARB1
EINT4_7	External interrupt 4 – 7	ARB1
EINT3	External interrupt 3	ARB0
EINT2	External interrupt 2	ARB0
EINT1	External interrupt 1	ARB0
EINT0	External interrupt 0	ARB0

由上图可得 EINT4_7 EINT8_23 合用一条中断线 ARB1

也就是可以直接到达 SRCPND 不需要设置 SUBSRCPND 和 SUBMASK 这两个寄存器 我们使用的外部中断源只需要设置 SRCPND MASK INTPND 这三个就可以




```
/* SRCPND 用来显示哪个中断产生了， 需要清除对应位，我们只需要关心
 * bit0 对应 eint0
 * bit2 对应 eint2
 * bit5 对应 eint8_23(表明 bit5 等于 1 的时候 eint8_23 中的某一个已经产生，我们需要继续分辨
 * 读 EINTPEND 分辨率哪个 EINT 产生)
 */
```

INTMOD 寄存器 默认值为 IRQ 模式即可，不需要设置

Register	Address	R/W	Description	Reset Value
INTMOD	0X4A000004	R/W	Interrupt mode register. 0 = IRQ mode 1 = FIQ mode	0x00000000

INTMASK 寄存器，需要设置为 0

Register	Address	R/W	Description	Reset Value
INTMSK	0X4A000008	R/W	Determine which interrupt source is masked. The masked interrupt source will not be serviced. 0 = Interrupt service is available. 1 = Interrupt service is masked.	0xFFFFFFFF

/* INTMSK 用来屏蔽中断, 1 对应 masked 屏蔽中断，我们需要设置相应位设置为 0
 * bit0-eint0
 * bit2-eint2
 * bit5-eint8_23
 */

同时可能有多个中断产生，这么多个中断经过优先级以后，只会有一个通知 CPU，是哪一个中断优先级最高，可以读 INTPAD 就能知道当前处理的唯 一 一个中断是那一个

Register	Address	R/W	Description	Reset Value
INTPND	0X4A000010	R/W	Indicate the interrupt request status. 0 = The interrupt has not been requested. 1 = The interrupt source has asserted the interrupt request.	0x00000000

1 表示这个中断已经产生 需要配置相应的位

/* INTPND 用来显示当前优先级最高的、正在发生的中断, 需要清除对应位
 * bit0-eint0
 * bit2-eint2
 * bit5-eint8_23
 */

INTOFFSET 是用来显示 INTPND 寄存器中哪一位正在等待处理

Register	Address	R/W	Description	Reset Value
INTOFFSET	0x4A000014	R	Indicate the IRQ interrupt request source	0x00000000

INT Source	The OFFSET value	INT Source	The OFFSET value
INT_ADC	31	INT_UART2	15
INT_RTC	30	INT_TIMER4	14
INT_SPI1	29	INT_TIMER3	13
INT_UART0	28	INT_TIMER2	12
INT_IIC	27	INT_TIMER1	11
INT_USBH	26	INT_TIMER0	10
INT_USBD	25	INT_WDT_AC97	9
INT_NFCON	24	INT_TICK	8
INT_UART1	23	nBATT_FLT	7
INT_SPI0	22	INT_CAM	6
INT_SDI	21	EINT8_23	5
INT_DMA3	20	EINT4_7	4
INT_DMA2	19	EINT3	3
INT_DMA1	18	EINT2	2
INT_DMA0	17	EINT1	1
INT_LCD	16	EINT0	0

INTPAD 中 bit0 等于 1 的话 INTOFFSET 就等于 0 INTPAD 中 bit1 等于 1 的话 INTOFFSET 值就等于 1 /* INTOFFSET : 用来显示 INTPND 中哪一位被设置为 1

*/

SRCPND 我们用不到

Register	Address	R/W	Description	Reset Value
SUBSRCPND	0X4A000018	R/W	Indicate the interrupt request status. 0 = The interrupt has not been requested. 1 = The interrupt source has asserted the interrupt request.	0x00000000

SUBSRCPND	Bit	Description	Initial State
Reserved	[31:15]	Not used	0
INT_AC97	[14]	0 = Not requested, 1 = Requested	0
INT_WDT	[13]	0 = Not requested, 1 = Requested	0
INT_CAM_P	[12]	0 = Not requested, 1 = Requested	0
INT_CAM_C	[11]	0 = Not requested, 1 = Requested	0
INT_ADC_S	[10]	0 = Not requested, 1 = Requested	0
INT_TC	[9]	0 = Not requested, 1 = Requested	0
INT_ERR2	[8]	0 = Not requested, 1 = Requested	0
INT_TXD2	[7]	0 = Not requested, 1 = Requested	0
INT_RXD2	[6]	0 = Not requested, 1 = Requested	0
INT_ERR1	[5]	0 = Not requested, 1 = Requested	0
INT_TXD1	[4]	0 = Not requested, 1 = Requested	0
INT_RXD1	[3]	0 = Not requested, 1 = Requested	0
INT_ERR0	[2]	0 = Not requested, 1 = Requested	0
INT_TXD0	[1]	0 = Not requested, 1 = Requested	0
INT_RXD0	[0]	0 = Not requested, 1 = Requested	0

Map To SRCPND

SRCPND	SUBSRCPND	Remark
INT_UART0	INT_RXD0,INT_TXD0,INT_ERR0	
INT_UART1	INT_RXD1,INT_TXD1,INT_ERR1	
INT_UART2	INT_RXD2,INT_TXD2,INT_ERR2	
INT_ADC	INT_ADC_S, INT_TC	
INT_CAM	INT_CAM_C, INT_CAM_P	
INT_WDT_AC97	INT_WDT, INT_AC97	

某一位等于 1 时 INT_UART0 它的来源可能有多个，这是串口 0 的中断，串口 0 的中断产生时有可能是接收到了数据(INT_RXD0),有可能是发送了数据(INT_TXD0),也有可能是产生了错误，那么到底是哪一个呢？ 需要去读取 SUBSRCPND 下一级的源寄存器

我们只需要设置 INTMSK 这个寄存器

SRCPND 和 INTPND 只有发生中断才需要设置

/* 初始化中断控制器 */ void interrupt_init(void) { //1 是屏蔽我们需要清零，外部中断 0 外部中断 2 外部中 8_23 里面还有外部中断 11 到 19 INTMSK &= ~((1<<0) | (1<<2) | (1<<5));

}

修改 start.S 删除 bl interrupt_init /* 初始化中断控制器 */ bl key_eint_init /* 初始化按键, 设为中断源 */

能使用 c 语言就使用 C 语言 在 main.c 文件中添加调用 C 函数

```
int main(void)
{
/************1/

   interrupt_init();  /* 初始化中断控制器 */

   key_eint_init();   /* 初始化按键, 设为中断源 */

*******/
   puts("\n\rg_A = ");
   printHex(g_A);
   puts("\n\r");
```

第 007 节_按键中断程序示例_完善

首先 main.c 中 我们初始化中断控制器 初始化中断源

假设按键按键就会产生中断，CPU 就会跳到 start.S 执行

```
_start:
   b reset          /* vector 0 : reset */
   ldr pc, und_addr /* vector 4 : und */
   ldr pc, swi_addr /* vector 8 : swi */
```

具体跳到哪里执行，我们需要看看中断向量表在哪里

Table 2-3. Exception Vectors

Address	Exception	Mode in Entry
0x00000000	Reset	Supervisor
0x00000004	Undefined instruction	Undefined
0x00000008	Software Interrupt	Supervisor
0x0000000C	Abort (prefetch)	Abort
0x00000010	Abort (data)	Abort
0x00000014	Reserved	Reserved
0x00000018	IRQ	IRQ
0x0000001C	FIQ	FIQ

IRQ 模式的话跳到 0x00000018 地方

```
b halt                    /* vector 0x0c  : prefetch aboot */
b halt                    /* vector 0x10  : data abort */
b halt                    /* vector 0x14  : reserved */
ldr pc, irq_addr /* vector 0x18  : irq */
b halt                    /* vector 0x1c  : fiq */
```

/*************3/ do_irq: /* 执行到这里之前：* 1. lr_irq 保存有被中断模式中的下一条即将执行的指令的地址 * 2. SPSR_irq 保存有被中断模式的 CPSR * 3. CPSR 中的 M4-M0 被设置为 10010, 进入到 irq 模式 * 4. 跳到 0x18 的地方执行程序 */

/* sp_irq 未设置，先设置它 */ /************ 分配不冲突的没有使用的内存就可以了 4*/ ldr sp, =0x33d00000

/************5

- /

/* 保存现场 */ /***********7 发生中断时 irq 返回值是 R14 -4 为什么要减去 4，硬件结构让你怎么做就怎么做

- /

/* 在 irq 异常处理函数中有可能会修改 r0-r12, 所以先保存 */ /* lr-4 是异常处理完后的返回地址, 也要保存 */ sub lr, lr, #4 stmdb sp!, {r0-r12, lr}

/* 处理 irq 异常 */ /***********6 在这 C 函数里分辨中断源，处理中断

- /

bl handle_irq_c

/***********8*/ /* 恢复现场 */ ldmia sp!, {r0-r12, pc}^ /* ^会把 spsr_irq 的值恢复到 cpsr 里 */

接下来我们在 interrupt.c 中写出 handle_irq_c 处理函数 这个是处理中断的 C 函数

```
    void handle_irq_c(void)
    {
        /*1 分辨中断源 */

        /*读 INTOFFSET 在芯片手册里找到这个寄存器，它里面的值表示
INTPND 中哪一位被设置成 1*/
        int bit = INTOFFSET;

        /*2 调用对应的处理函数 */

        if (bit == 0 || bit == 2 || bit == 5)  /* 对应
eint0,2,eint8_23 */
        {
                /*我们会调用一个按键处理函数*/

                key_eint_irq(bit); /* 处理中断，清中断源
EINTPEND */
        }

        /*3 清中断 : 从源头开始清

         *先清除掉中断源里面的某些寄存器

         *再清 SRCPND

         *再清 INTPND

         */
        SRCPND = (1<<bit);
        INTPND = (1<<bit);
    }
```

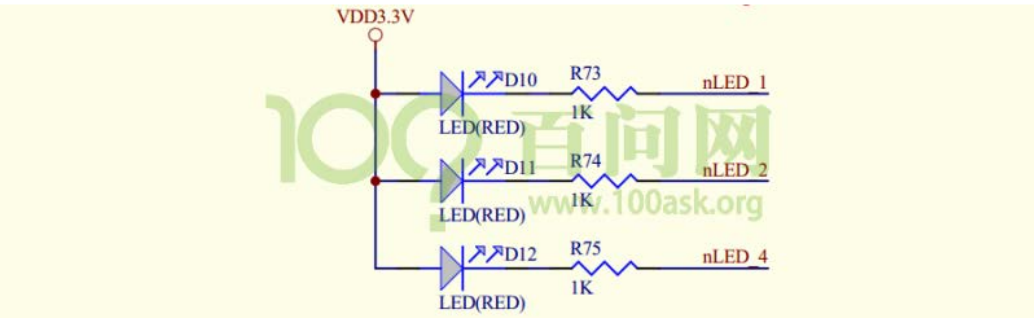
```
/* 读 EINTPEND 分辨率哪个 EINT 产生(eint4~23)

    * 清除中断时，写 EINTPEND 的相应位
    */
```

```
void key_eint_irq(int irq)
{
    /**4 清的时候我先把这个值读出来，清的时候我再把他写进去/
    unsigned int val = EINTPEND;

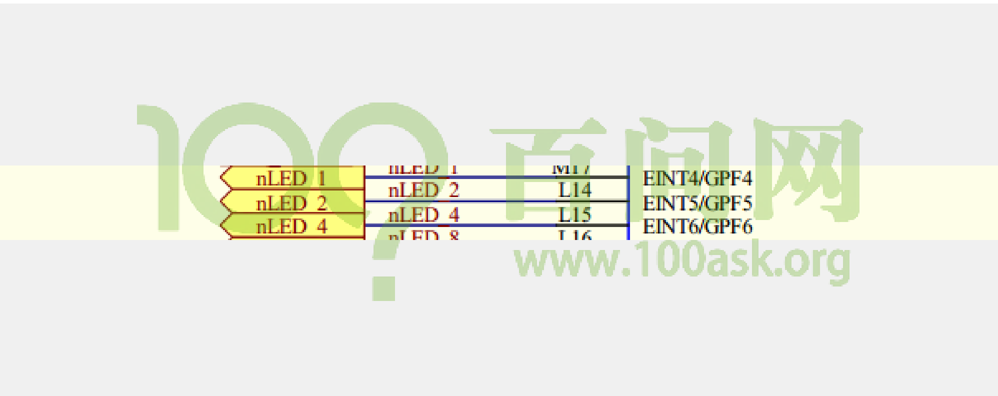
    unsigned int val1 = GPFDAT;
    unsigned int val2 = GPGDAT;

    if (irq == 0) /*1 外部中断eint0 对应s2 按键  */
    {
```



/*1.1 我们使用 s2 来控制那盏灯？

- 之前我们写过按键控制 led 灯的程序，它使用的是 s2 控制 gpf6
- 也就是 s2 控制 led4 D12




```
        */
        /****1.2 如何知道是按下还是松开，我们需要读值*/
        if (val1 & (1<<0)) /* s2 --> gpf6 */
        {
```

```
                /*1.3 松开 */
                GPFDAT |= (1<<6);
            }
            else
            {
                /*1.4 按下 */
                GPFDAT &= ~(1<<6);
            }

        }
        else if (irq == 2) /*2 eint2 对应 s3 控制 D11  LED2 */
        {
            if (val1 & (1<<2)) /* s3 --> gpf5 */
            {
                /* 松开 */
                GPFDAT |= (1<<5);
            }
            else
            {
                /* 按下 */
                GPFDAT &= ~(1<<5);
            }

        }
        else if (irq == 5) /*3 eint8_23, eint11 对应 s4 控制
D10 LED1, eint19 对应 s5 控制所有 LED */
        {
```

/*3.1 到底是发生哪一种中断，我们需要读取 EINTPND 来判断是那个中断

EINTPEND (External Interrupt Pending Register)

Register	Address	R/W	Description	Reset Value
EINTPEND	0x560000a8	R/W	External interrupt pending register	0x00

EINTPEND	Bit	Description	Reset Value
EINT23	[23]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
EINT22	[22]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
EINT21	[21]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
EINT20	[20]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
EINT19	[19]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
EINT18	[18]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
EINT17	[17]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
EINT16	[16]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
EINT15	[15]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
EINT14	[14]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
EINT13	[13]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
EINT12	[12]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
EINT11	[11]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
EINT10	[10]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
EINT9	[9]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
EINT8	[8]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
EINT7	[7]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
EINT6	[6]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
EINT5	[5]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
EINT4	[4]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
Reserved	[3:0]	Reserved	0000

/*如果 bit19 等于 1 的话表明外部中断 EINT19 产生了，如果 bit11 等于 1 表用外部中断 11 产生，这里我们需要判断*/

```
if (val & (1<<11)) /* 表明外部中断eint11 产生 */
{
        if (val2 & (1<<3)) /* s4 --> gpf4 */
        {
                /* 松开 */
                GPFDAT |= (1<<4);
        }
        else
        {
                /* 按下 */
                GPFDAT &= ~(1<<4);
```

```
				}
			}
			else if (val & (1<<19)) /* 表用外部中断eint19
*/
			{
				if (val2 & (1<<11))
				{
					/* 松开 */

					/* 熄灭所有LED 输出高电平 */

					GPFDAT |= ((1<<4) | (1<<5) |
(1<<6));
				}
				else
				{
					/* 按下: 点亮所有LED */

					GPFDAT &= ~((1<<4) | (1<<5) |
(1<<6));
				}
			}
		}
		/**5 再把值写进去就达到了清除的效果*/

		EINTPEND = val;
	}
```

上传代码测试

我们需要包含头文件

#include "S3c2440_soc.h"

编译通过，开发板上电测试发现按键 s5 无法控制 查看 interrupt.c 文件中的按键初始化

```
	/* 初始化按键, 设为中断源 */

	void key_eint_init(void)
	{
		/* 配置GPIO为中断引脚 */

		GPFCON &= ~((3<<0) | (3<<4));
```

```
    GPFCON |= ((2<<0) | (2<<4));   /* S2,S3 被配置为中断引
脚 */
```

发现外部中断 19 的 bit 位配置不正确应该是 22

```
![](https://i.imgur.com/EWZaHFx.png)
    GPGCON &= ~((3<<6) | (3<<22));

    GPGCON |= ((2<<6) | (2<<22));   /* S4,S5 被配置为中断引
脚 */
```

上传代码从新编译执行 重新烧写看是否可以使用 回顾中断处理流程
我们 start.s 一上电从 _start: 运行 做一些初始化工作

```
reset:
    /* 关闭看门狗 */
    ldr r0, =0x53000000
    ldr r1, =0
    str r1, [r0]

    /* 设置 MPLL, FCLK : HCLK : PCLK = 400m : 100m : 50m
*/
    /* LOCKTIME(0x4C000000) = 0xFFFFFFFF */
    ldr r0, =0x4C000000
    ldr r1, =0xFFFFFFFF
    str r1, [r0]

    /* CLKDIVN(0x4C000014) = 0X5, tFCLK:tHCLK:tPCLK =
1:4:8  */
    ldr r0, =0x4C000014
    ldr r1, =0x5
    str r1, [r0]

    /* 设置 CPU 工作于异步模式 */

    mrc p15,0,r0,c1,c0,0
    orr r0,r0,#0xc0000000   //R1_nF:OR:R1_iA
    mcr p15,0,r0,c1,c0,0

    /* 设置 MPLLCON(0x4C000004) = (92<<12)|(1<<4)|(1<<0)
```

```
 *  m = MDIV+8 = 92+8=100
 *  p = PDIV+2 = 1+2 = 3
 *  s = SDIV = 1
 *  FCLK = 2*m*Fin/(p*2^s) = 2*100*12/(3*2^1)=400M
 */
ldr r0, =0x4C000004
ldr r1, =(92<<12)|(1<<4)|(1<<0)
str r1, [r0]

/* 一旦设置PLL, 就会锁定lock time直到PLL输出稳定

 * 然后CPU工作于新的频率FCLK

 */




/* 设置内存: sp 栈 */

/* 分辨是nor/nand启动

 * 写0到0地址, 再读出来

 * 如果得到0, 表示0地址上的内容被修改了, 它对应ram, 这就

是nand启动

 * 否则就是nor启动
 */
mov r1, #0

ldr r0, [r1] /* 读出原来的值备份 */

str r1, [r1] /* 0->[0] */
ldr r2, [r1] /* r2=[0] */

cmp r1, r2   /* r1==r2? 如果相等表示是NAND启动 */

ldr sp, =0x40000000+4096 /* 先假设是nor启动 */

moveq sp, #4096  /* nand启动 */

streq r0, [r1]   /* 恢复原来的值 */
```

```
        bl sdram_init
        //bl sdram_init2    /* 用到有初始值的数组, 不是位置无关码
*/

        /* 重定位 text, rodata, data 段整个程序 */
        bl copy2sdram

        /* 清除 BSS 段 */
        bl clean_bss

        /* 复位之后, cpu 处于 svc 模式
         * 现在, 切换到 usr 模式
         */
        mrs r0, cpsr        /* 读出 cpsr */

        bic r0, r0, #0xf    /* 修改 M4-M0 为 0b10000, 进入 usr
模式 */

        bic r0, r0, #(1<<7)  /* 清除 I 位, 使能中断 */
        msr cpsr, r0

        /* 设置 sp_usr */
        ldr sp, =0x33f00000

        ldr pc, =sdram
    sdram:
        bl uart0_init

        bl print1
        /* 故意加入一条未定义指令 */
    und_code:
        .word 0xdeadc0de  /* 未定义指令 */
        bl print2
```

```
    swi 0x123  /* 执行此命令, 触发SWI 异常, 进入 0x8 执行 */

    //bl main  /* 使用BL 命令相对跳转, 程序仍然在NOR/sram 执
行 */

    ldr pc, =main  /* 绝对跳转, 跳到SDRAM */

halt:
    b halt
```

让后设置 CPSR 开中断

让后调到 mian 函数，做一些中断初始化

```
int main(void)
{
    led_init();

    interrupt_init();  /* 初始化中断控制器 */

    key_eint_init();   /* 初始化按键, 设为中断源 */

    puts("\n\rg_A = ");
    printHex(g_A);
    puts("\n\r");

/*****让后在main 函数里一直循环输出串口*/
    while (1)
    {

          putchar(g_Char);
          g_Char++;

          putchar(g_Char3);
          g_Char3++;
          delay(1000000);
    }
```

这个时候按下按键就会产生中断，让后进入 start.s

跳到 0x18 irq 模式

```
ldr pc, irq_addr /* vector 0x18 : irq */
```

它是一条读内存的执行，从这里读地址赋给 pc

```
irq_addr:
    .word do_irq
```

就跳到 sdram 执行 do_irq 函数

```
do_irq:
    /* 执行到这里之前:
     * 1. lr_irq 保存有被中断模式中的下一条即将执行的指令的地址
     * 2. SPSR_irq 保存有被中断模式的 CPSR
     * 3. CPSR 中的 M4-M0 被设置为 10010, 进入到 irq 模式
     * 4. 跳到 0x18 的地方执行程序
     */

    /* sp_irq 未设置, 先设置它 */
    ldr sp, =0x33d00000

    /* 保存现场 */
    /* 在 irq 异常处理函数中有可能会修改 r0-r12, 所以先保存 */
    /* lr-4 是异常处理完后的返回地址, 也要保存 */
    sub lr, lr, #4
    stmdb sp!, {r0-r12, lr}

    /* 处理 irq 异常 */
    bl handle_irq_c

    /* 恢复现场 */
```

它怎么处理

```
/* 读 EINTPEND 分辨率哪个 EINT 产生(eint4~23)
 * 清除中断时, 写 EINTPEND 的相应位
```

```
 */


void key_eint_irq(int irq)
{
    unsigned int val = EINTPEND;
    unsigned int val1 = GPFDAT;
    unsigned int val2 = GPGDAT;

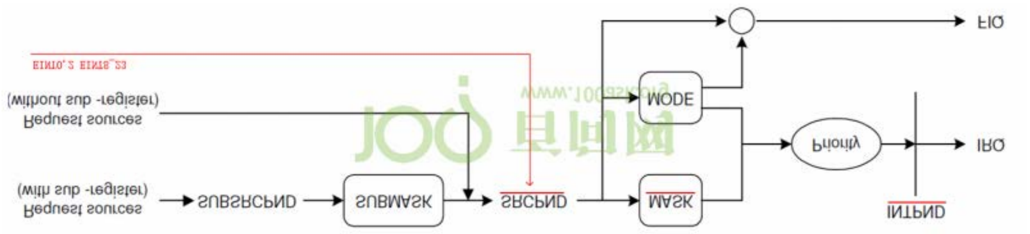
    if (irq == 0) /* eint0 : s2 控制 D12 */
    {
            if (val1 & (1<<0)) /* s2 --> gpf6 */
            {
                    /* 松开 */
                    GPFDAT |= (1<<6);
            }
            else
            {
                    /* 按下 */
                    GPFDAT &= ~(1<<6);
            }

    }
    else if (irq == 2) /* eint2 : s3 控制 D11 */
    {
            if (val1 & (1<<2)) /* s3 --> gpf5 */
            {
                    /* 松开 */
                    GPFDAT |= (1<<5);
            }
            else
            {
                    /* 按下 */
                    GPFDAT &= ~(1<<5);
            }

    }
```

```
	else if (irq == 5) /* eint8_23, eint11--s4 控制 D10,
eint19---s5 控制所有 LED */
	{
		if (val & (1<<11)) /* eint11 */
		{
			if (val2 & (1<<3)) /* s4 --> gpf4 */
			{
				/* 松开 */
				GPFDAT |= (1<<4);
			}
			else
			{
				/* 按下 */
				GPFDAT &= ~(1<<4);
			}
		}
		else if (val & (1<<19)) /* eint19 */
		{
			if (val2 & (1<<11))
			{
				/* 松开 */
				/* 熄灭所有 LED */
				GPFDAT |= ((1<<4) | (1<<5) |
(1<<6));
			}
			else
			{
				/* 按下: 点亮所有 LED */
				GPFDAT &= ~((1<<4) | (1<<5) |
(1<<6));
			}
		}
	}
```

EINTPEND = val; } 处理完之后清中断，从源头开始清 这完全是按照中断流程操作的



第 008 节_定时器中断程序示例

这节课我们来写一个定时器的中断服务程序 使用定时器来实现点灯计数 查考资料就是第 10 章 PWM TIMER 可以参考书籍《嵌入式 Linux 应用程序开发完全手册》第 10 章

我们先把这个结构图展示出来

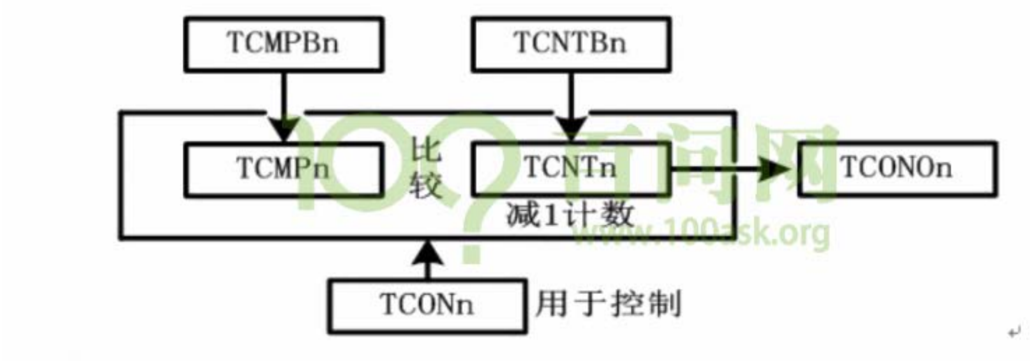


图 10.4 定时器内部控制逻辑图

这个图的结构很好

这里面肯定有一个 clk(时钟)， 1 每来一个 clk(时钟)这个 TCNTn 减去 1 2 当 TCNTn == TCMPn 时，可以产生中断，也可以让对应的 SPWM 引脚反转，(比如说原来是高电平，发生之后电平转换成低电平) 3 TCNTn 继续减 1，当 TCNTn == 0 时，可以产生中断，pwm 引脚再次反转 TCMPn 和 TCNTn 的初始值来自 TCMPBn,TCNTBn 4 TCNTn == 0 时，可自动加载初始

怎么使用定时器？

1 设置时钟
2 设置初值
3 加载初始，启动 Timer
4 设置为自动加载
5 中断相关

由于 2440 没有引出 pwm 引脚，所以 pwm 功能无法使用，也就无法做 pwm 相关实验，所谓 pwm 是指可调制脉冲



T1 高脉冲和 T2 低脉冲它的时间 T1，T2 可调整，可以输出不同频率不同占控比的波形，在控制电机时特别有用

我们这个程序只做一个实验，当 TCNTn 这个计数器到 0 的时候，就产生中断，在这个中断服务程序里我们点灯

写代码 打开我们的 main 函数

```
int main(void)
{
    led_init();

    interrupt_init();  /* 初始化中断控制器 */

//我们初始化了中断源，同样的，我们初始化 timer

    key_eint_init();   /* 初始化按键，设为中断源 */

//初始化定时器

    timer_init();
```

我们需要实现定时器初始化函数

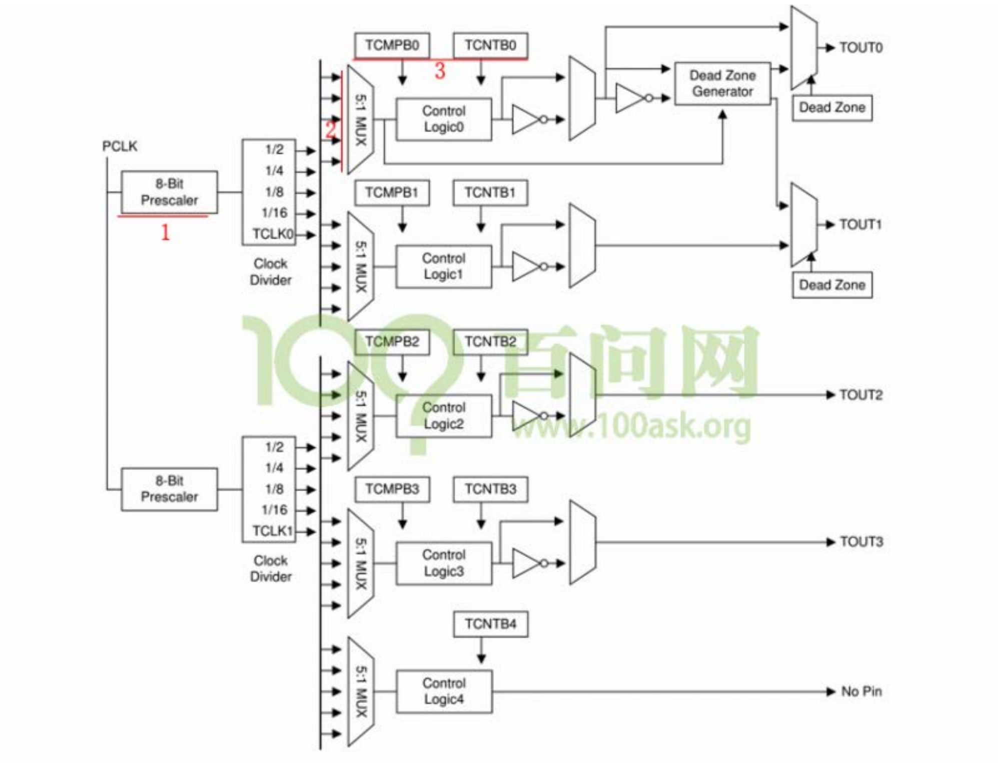
新建一个 timer.c ，我们肯定需要操作一堆寄存器，添加头文件

```
#include "s3c2440_soc.h"
void timer_init(void)
{
```

/*1 设置 TIMER0 的时钟 */ /*2 设置 TIMER0 的初值 */ /*3 加载初值，启动 timer0 */ /*4 设置为自动加载并启动(值到 0 以后会自动加载) */ /*5 设置中断，显然我们需要提供一个中断处理函数 void timer_irq(void) *在这里面我们需要点灯*/

打开芯片手册，我们想设置 timer0 的话

1 首先设置 8-Bit Prescaler
2 设置 5.1 MUX(选择一个时钟分频)
3 设置 TCMPB 0 和 TCNTB0
4 设置 TCONn 寄存器



看手册上写如何初始化 timer

TIMER INITIALIZATION USING MANUAL UPDATE BIT AND INVERTER BIT

An auto reload operation of the timer occurs when the down counter reaches 0. So, a starting value of the TCNTn has to be defined by the user in advance. In this case, the starting value has to be loaded by the manual update bit. The following steps describe how to start a timer:

1) Write the initial value into TCNTBn and TCMPBn.
2) Set the manual update bit of the corresponding timer. It is recommended that you configure the inverter on/off bit. (Whether use inverter or not).
3) Set start bit of the corresponding timer to start the timer (and clear the manual update bit).

If the timer is stopped by force, the TCNTn retains the counter value and is not reloaded from TCNTBn. If a new value has to be set, perform manual update.

1 把初始值写到 TCNTBn 和 TCMPBn 寄存器 2 设置手动更新位 3 设置启动位

往下看到时钟配置寄存器

TIMER CONFIGURATION REGISTER0 (TCFG0)

Timer input clock Frequency = PCLK / {prescaler value+1} / {divider value}
{prescaler value} = 0~255
{divider value} = 2, 4, 8, 16

Register	Address	R/W	Description	Reset Value
TCFG0	0x51000000	R/W	Configures the two 8-bit prescalers	0x00000000

TCFG0	Bit	Description	Initial State
Reserved	[31:24]		0x00
Dead zone length	[23:16]	These 8 bits determine the dead zone length. The 1 unit time of the dead zone length is equal to that of timer 0.	0x00
Prescaler 1	[15:8]	These 8 bits determine prescaler value for Timer 2, 3 and 4.	0x00
Prescaler 0	[7:0]	These 8 bits determine prescaler value for Timer 0 and 1.	0x00

有个计算公式

Timer clk = PCLK / {(预分频数)prescaler value+1} / {divider value(5.1MUX 值)}

PCLK 是 50M = 50000000/(99+1)/16 = 31250 也就是说我们得 TCON 是 31250 的话，从这个值一直减到 0
Prescaler0 等于 99

TCFG0 = 99; /* Prescaler 0 = 99, 用于 timer0,1 */

TCFG1 MUX 多路复用器的意思，他有多路输入，我们可以通过 MUX 选择其中一路作为输出

TIMER CONFIGURATION REGISTER1 (TCFG1)

Register	Address	R/W	Description	Reset Value
TCFG1	0x51000004	R/W	5-MUX & DMA mode selection register	0x00000000

TCFG1	Bit	Description	Initial State
Reserved	[31:24]		00000000
DMA mode	[23:20]	Select DMA request channel 0000 = No select (all interrupt) 0001 = Timer0 0010 = Timer1 0011 = Timer2 0100 = Timer3 0101 = Timer4 0110 = Reserved	0000
MUX 4	[19:16]	Select MUX input for PWM Timer4. 0000 = 1/2 0001 = 1/4 0010 = 1/8 0011 = 1/16 01xx = External TCLK1	0000
MUX 3	[15:12]	Select MUX input for PWM Timer3. 0000 = 1/2 0001 = 1/4 0010 = 1/8 0011 = 1/16 01xx = External TCLK1	0000
MUX 2	[11:8]	Select MUX input for PWM Timer2. 0000 = 1/2 0001 = 1/4 0010 = 1/8 0011 = 1/16 01xx = External TCLK1	0000
MUX 1	[7:4]	Select MUX input for PWM Timer1. 0000 = 1/2 0001 = 1/4 0010 = 1/8 0011 = 1/16 01xx = External TCLK0	0000
MUX 0	[3:0]	Select MUX input for PWM Timer0. 0000 = 1/2 0001 = 1/4 0010 = 1/8 0011 = 1/16 01xx = External TCLK0	0000

根据上面 mux 的值，我们要把 MUX0 设置成 0011 只需要设置这 4 位即可，先清零 再或上 0011 就是 3

```
TCFG1 &= ~0xf;
TCFG1 |= 3;  /* MUX0  : 1/16 */
```

再来看看初始值控制寄存器

TCNTB0	Bit	Description	Initial State
Timer 0 count buffer register	[15:0]	Set count buffer value for Timer 0	0x00000000

一秒钟点灯太慢了 ，就让 0.5 秒 TCNTB0 = 15625; /* 0.5s 中断一次 */
这个寄存器是用来观察里面的计数值的，不需要设置

现在可以设置 TCON 来设置这个寄存器了

TIMER CONTROL (TCON) REGISTER

Register	Address	R/W	Description	Reset Value
TCON	0x51000008	R/W	Timer control register	0x00000000

TCON	Bit	Description	Initial state
Timer 4 auto reload on/off	[22]	Determine auto reload on/off for Timer 4. 0 = One-shot 1 = Interval mode (auto reload)	0
Timer 4 manual update (note)	[21]	Determine the manual update for Timer 4. 0 = No operation 1 = Update TCNTB4	0
Timer 4 start/stop	[20]	Determine start/stop for Timer 4. 0 = Stop 1 = Start for Timer 4	0
Timer 3 auto reload on/off	[19]	Determine auto reload on/off for Timer 3. 0 = One-shot 1 = Interval mode (auto reload)	0
Timer 3 output inverter on/off	[18]	Determine output inverter on/off for Timer 3. 0 = Inverter off 1 = Inverter on for TOUT3	0
Timer 3 manual update (note)	[17]	Determine manual update for Timer 3. 0 = No operation 1 = Update TCNTB3 & TCMPB3	0
Timer 3 start/stop	[16]	Determine start/stop for Timer 3. 0 = Stop 1 = Start for Timer 3	0
Timer 2 auto reload on/off	[15]	Determine auto reload on/off for Timer 2. 0 = One-shot 1 = Interval mode (auto reload)	0
Timer 2 output inverter on/off	[14]	Determine output inverter on/off for Timer 2. 0 = Inverter off 1 = Inverter on for TOUT2	0
Timer 2 manual update (note)	[13]	Determine the manual update for Timer 2. 0 = No operation 1 = Update TCNTB2 & TCMPB2	0
Timer 2 start/stop	[12]	Determine start/stop for Timer 2. 0 = Stop 1 = Start for Timer 2	0
Timer 1 auto reload on/off	[11]	Determine the auto reload on/off for Timer1. 0 = One-shot 1 = Interval mode (auto reload)	0
Timer 1 output inverter on/off	[10]	Determine the output inverter on/off for Timer1. 0 = Inverter off 1 = Inverter on for TOUT1	0
Timer 1 manual update (note)	[9]	Determine the manual update for Timer 1. 0 = No operation 1 = Update TCNTB1 & TCMPB1	0
Timer 1 start/stop	[8]	Determine start/stop for Timer 1. 0 = Stop 1 = Start for Timer 1	0

现在需要设置 Timer0

TCON	Bit	Description	Initial state
Reserved	[7:5]	Reserved	
Dead zone enable	[4]	Determine the dead zone operation. 0 = Disable 1 = Enable	0
Timer 0 auto reload on/off	[3]	Determine auto reload on/off for Timer 0. 0 = One-shot 1 = Interval mode(auto reload)	0
Timer 0 output inverter on/off	[2]	Determine the output inverter on/off for Timer 0. 0 = Inverter off 1 = Inverter on for TOUT0	0
Timer 0 manual update (note)	[1]	Determine the manual update for Timer 0. 0 = No operation 1 = Update TCNTB0 & TCMPB0	0
Timer 0 start/stop	[0]	Determine start/stop for Timer 0. 0 = Stop 1 = Start for Timer 0	0

开始需要手工更新 TCON |= (1<<1); /* Update from TCNTB0 & TCMPB0 */ 把这两个值放到 TCNTB0 和 TCMPB0 中

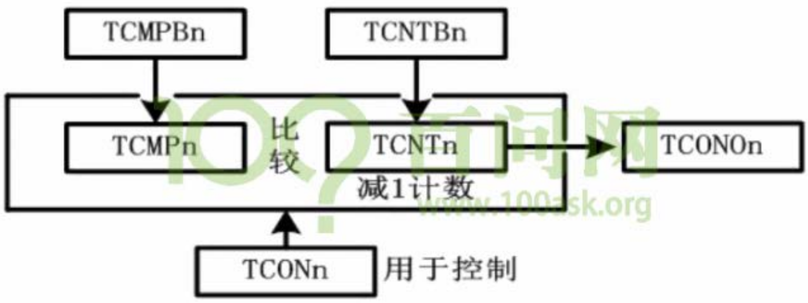


图 10.4 定时器内部控制逻辑图

注意：这一位必须清楚才能写下一位

设置为自动加载并启动，先清掉手动更新位，再或上 bit0 bit3

TCON &= (1<<1);
TCON |= (1<<0) | (1<<3); /* bit0: start, bit3: auto reload */

设置中断，显然我们需要提供一个中断处理函数 void timer_irq(void) 在 Timer 里没有看到中断相关的控制器，我们需要回到中断章节去看看中断控制器，看看有没有定时器相关的中断 我们没有看到更加细致的 Timer0 寄存器

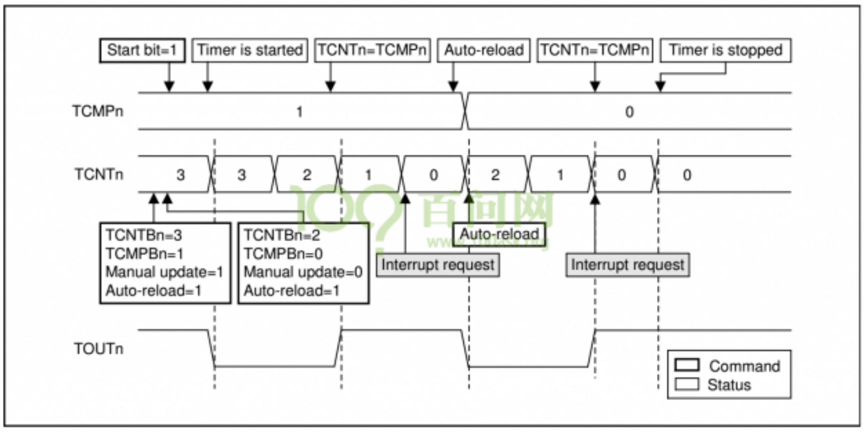


Figure 10-2. Timer Operations

当 TCNTn=TCMPn 时，他不会产生中断，只有当 TCNTn 等于 0 的时候才可以产生中断，我们之前以为这个定时器可以产生两种中断，那么肯定有寄存器中断或者禁止两种寄存器其中之一，那现在只有一种中断的话，就相对简单些 设置中断的话，我们只需要设置中断控制器

设置 interrupu.c 中断控制器 /* 初始化中断控制器 */ void interrupt_init(void) { INTMSK &= ~((1<<0) | (1<<2) | (1<<5)); //把定时器相应的位清零就可以了，哪一位呢？ INTPND 的哪一位？ INT_TIMER0 第 10 位即可

INTERRUPT PENDING (INTPND) REGISTER (Continued)

INTPND	Bit	Description	Initial State
INT_ADC	[31]	0 = Not requested, 1 = Requested	0
INT_RTC	[30]	0 = Not requested, 1 = Requested	0
INT_SPI1	[29]	0 = Not requested, 1 = Requested	0
INT_UART0	[28]	0 = Not requested, 1 = Requested	0
INT_IIC	[27]	0 = Not requested, 1 = Requested	0
INT_USBH	[26]	0 = Not requested, 1 = Requested	0
INT_USBD	[25]	0 = Not requested, 1 = Requested	0
INT_NFCON	[24]	0 = Not requested, 1 = Requested	0
INT_UART1	[23]	0 = Not requested, 1 = Requested	0
INT_SPI0	[22]	0 = Not requested, 1 = Requested	0
INT_SDI	[21]	0 = Not requested, 1 = Requested	0
INT_DMA3	[20]	0 = Not requested, 1 = Requested	0
INT_DMA2	[19]	0 = Not requested, 1 = Requested	0
INT_DMA1	[18]	0 = Not requested, 1 = Requested	0
INT_DMA0	[17]	0 = Not requested, 1 = Requested	0
INT_LCD	[16]	0 = Not requested, 1 = Requested	0
INT_UART2	[15]	0 = Not requested, 1 = Requested	0
INT_TIMER4	[14]	0 = Not requested, 1 = Requested	0
INT_TIMER3	[13]	0 = Not requested, 1 = Requested	0
INT_TIMER2	[12]	0 = Not requested, 1 = Requested	0
INT_TIMER1	[11]	0 = Not requested, 1 = Requested	0
INT_TIMER0	[10]	0 = Not requested, 1 = Requested	0

INTMSK &= ~(1<<10); /* enable timer0 int */ }

当定时器减到 0 的时候就会产生中断，就会进到 start.s 这里一路执行 do_irq

```
do_irq:
    /* 执行到这里之前:
     * 1. lr_irq 保存有被中断模式中的下一条即将执行的指令的地址
     * 2. SPSR_irq 保存有被中断模式的 CPSR
     * 3. CPSR 中的 M4-M0 被设置为 10010, 进入到 irq 模式
     * 4. 跳到 0x18 的地方执行程序
     */

    /* sp_irq 未设置, 先设置它 */
    ldr sp, =0x33d00000

    /* 保存现场 */
    /* 在 irq 异常处理函数中有可能会修改 r0-r12, 所以先保存 */
    /* lr-4 是异常处理完后的返回地址, 也要保存 */
    sub lr, lr, #4
    stmdb sp!, {r0-r12, lr}

    /* 处理 irq 异常 */
    bl handle_irq_c

    /* 恢复现场 */
    ldmia sp!, {r0-r12, pc}^  /* ^会把 spsr_irq 的值恢复到
cpsr 里 */
```

让后进入 irq 处理函数中处理，处理这个 irq

```
void handle_irq_c(void)
```

```
{
    /* 分辨中断源 */
    int bit = INTOFFSET;

    /* 调用对应的处理函数 */

/***************/

if(bit ==0 || bit == 2 || bit == 5)/*eint0,2,rint8_23*/
{
    key_eint_irq(bit);/*处理中断，清中断源EINTPEND*/

}else if(bit == 10)//如果等于10 的话说明发生的是定时器中断，
这时候就调用我们得timer_irq
{
    timer_irq();
}

    /* 清中断 : 从源头开始清 */
    SRCPND = (1<<bit);
    INTPND = (1<<bit);
}
```

回到timer.c文件中，在这个定时器处理函数中我们需要点灯

```
void timer_irq(void)
{
    /* 点灯计数 循环点灯*/
    static int cnt = 0;
    int tmp;

    cnt++;

    tmp = ~cnt;
    tmp &= 7;
    GPFDAT &= ~(7<<4);
    GPFDAT |= (tmp<<4);
```

```
}
```

代码写完我们实验一下，上传代码，在 MAKEFILE 中添加 timer.o，进行编译 编译后进行烧写 现象灯没有闪烁 他不是有一个观察寄存器么？我们不断的打印这个值，看是否有变化

TIMER 0 COUNT OBSERVATION REGISTER (TCNTO0)

Register	Address	R/W	Description	Reset Value
TCNTO0	0x51000014	R	Timer 0 count observation register	0x00000000

TCNTO0	Bit	Description	Initial State
Timer 0 observation register	[15:0]	Set count observation value for Timer 0	0x00000000

在 main 函数中不断打印

putchar(g_Char3); g_Char3++; delay(1000000); /************/ printHex(TCNTO0); } 编译实验 打印结果全都是 0，发现我们的定时器根本就没有启用，在 timer.c 文件 void timer_init(void)函数里

设置为自动加载并启动，先清掉手动更新位，再或上 bit0 bit3

TCON &= ~(1<<1);//我们没有设置取反 TCON |= (1<<0) | (1<<3); /* bit0: start, bit3: auto reload */ 再次实验 发现灯已经开始闪，就可以把调试信息去除了 对程序进行改进 进入 main 函数中执行 timer_init();

还需要修改 interrupt.c 初始化函数 void interrupt_init(void) 还需要调用中断处理函数 void handle_irq_c(void) 每次添加一个中断我都需要修改 handle_irq 这个函数，这样太麻烦，我能不能保证这个 interrupt 文件不变，只需要在 timer.c 中引用即可，这里我们使用指针数组

在 interrupt.c 中定义函数指针数组

typedef void(*irq_func)(int); 定义一个数组，我们来卡看下这里有多少项，一共 32 位，我们想把每一个中断的处理函数都放在这个数组里面来，当发生中断时，我们可以得到这个中断号，让后我从数组里面调用对应的中断号就可以了 irq_func irq_array[32];

那么我们得提供一个注册函数

```
void register_irq (int irq, irq_func fp)
{
    irq_array[irq] = fp;
    INTMASK &= ~(1 << irq)
}
```

以后我直接调用对应的处理函数

```
void handle_irq_c(void)
{
    /* 分辨中断源 */
    int bit = INTOFFSET;

/************/
```

```
/* 调用对应的处理函数 */
irq_array[bit](bit);

    /* 清中断 : 从源头开始清 */
    SRCPND = (1<<bit);
    INTPND = (1<<bit);
}
```

按键中断初始化函数需要注册

```
    /* 初始化按键, 设为中断源 */
    void key_eint_init(void)
    {
    /* 配置 GPIO 为中断引脚 */
    GPFCON &= ~((3<<0) | (3<<4));
    GPFCON |= ((2<<0) | (2<<4));   /* S2,S3 被配置为中断引
脚 */

    GPGCON &= ~((3<<6) | (3<<22));
    GPGCON |= ((2<<6) | (2<<22));   /* S4,S5 被配置为中断引
脚 */


    /* 设置中断触发方式: 双边沿触发 */
    EXTINT0 |= (7<<0) | (7<<8);     /* S2,S3 */
    EXTINT1 |= (7<<12);             /* S4 */
    EXTINT2 |= (7<<12);             /* S5 */

    /* 设置 EINTMASK 使能 eint11,19 */
    EINTMASK &= ~((1<<11) | (1<<19));

register_irq(0, key_eint_irq);
register_irq(2, key_eint_irq);
register_irq(5, key_eint_irq);
```

```
}
```

在 timer.c 中也需要设置中断

```
void timer_init(void)
{
/* 设置 TIMER0 的时钟 */
/* Timer clk = PCLK / {prescaler value+1} / {divider value}
              = 50000000/(99+1)/16
              = 31250
 */
TCFG0 = 99;  /* Prescaler 0 = 99, 用于 timer0,1 */
TCFG1 &= ~0xf;
TCFG1 |= 3;  /* MUX0 : 1/16 */

/* 设置 TIMER0 的初值 */
TCNTB0 = 15625;  /* 0.5s 中断一次 */

/* 加载初值, 启动 timer0 */
TCON |= (1<<1);   /* Update from TCNTB0 & TCMPB0 */

/* 设置为自动加载并启动 */
TCON &= ~(1<<1);
TCON |= (1<<0) | (1<<3);  /* bit0: start, bit3: auto reload */

/* 设置中断 */
register_irq(10, timer_irq);
}
```

把 interrupt.c 中按键的初始化放在最后面

我们来看看我们做了什么事情， 1 我们定义了一个指针数组 typedef void(*irq_func)(int); 这个指针数组里面放有各个指针的处理函数 irq_func irq_array[32]; 当我们去初始化按键中断时，我们给这按键注册中断函数

```
register_irq(0, key_eint_irq);
```

```
register_irq(2, key_eint_irq);
register_irq(5, key_eint_irq);
```

这个注册函数会做什么事情，他会把这个数组放在注册函数里面，同时使能中断

```
void register_irq(int irq, irq_func fp)
{
    irq_array[irq] = fp;

    INTMSK &= ~(1<<irq);
}
```

我们的 timer.c 中

```
timer_init();
```

也会注册这个函数

```
    /* 设置中断 */
    register_irq(10, timer_irq);
```

把这个中断 irq 放在第 10 项里同时使能中断，以后我们只需要添加中断号，和处理函数即可，再也不需要修改函数 烧写执行

我们从 start.s 开始看， 一上电从 b reset 运行做一列初始化

```
.text
.global _start

_start:
    b reset          /* vector 0 : reset */
    ldr pc, und_addr /* vector 4 : und */
    ldr pc, swi_addr /* vector 8 : swi */
    b halt              /* vector 0x0c : prefetch aboot
*/
    b halt              /* vector 0x10 : data abort */
    b halt              /* vector 0x14 : reserved */


reset:
    /* 关闭看门狗 */
    ldr r0, =0x53000000
    ldr r1, =0
    str r1, [r0]
```

```
	/* 设置MPLL, FCLK : HCLK : PCLK = 400m : 100m : 50m 
*/
	/* LOCKTIME(0x4C000000) = 0xFFFFFFFF */
	ldr r0, =0x4C000000
	ldr r1, =0xFFFFFFFF
	str r1, [r0]

	/* CLKDIVN(0x4C000014) = 0X5, tFCLK:tHCLK:tPCLK = 
1:4:8  */
	ldr r0, =0x4C000014
	ldr r1, =0x5
	str r1, [r0]


	/* 设置CPU工作于异步模式 */

	mrc p15,0,r0,c1,c0,0
	orr r0,r0,#0xc0000000   //R1_nF:OR:R1_iA
	mcr p15,0,r0,c1,c0,0


	/* 设置MPLLCON(0x4C000004) = (92<<12)|(1<<4)|(1<<0) 

	 *  m = MDIV+8 = 92+8=100
	 *  p = PDIV+2 = 1+2 = 3
	 *  s = SDIV = 1
	 *  FCLK = 2*m*Fin/(p*2^s) = 2*100*12/(3*2^1)=400M
	 */
	ldr r0, =0x4C000004
	ldr r1, =(92<<12)|(1<<4)|(1<<0)
	str r1, [r0]


	/* 一旦设置PLL, 就会锁定lock time直到PLL输出稳定

	 * 然后CPU工作于新的频率FCLK

	 */




	/* 设置内存: sp 栈 */

	/* 分辨是nor/nand启动

	 * 写0到0地址, 再读出来
```

```
	 * 如果得到0，表示0地址上的内容被修改了，它对应ram，这就是nand启动
	 * 否则就是nor启动
	 */
	mov r1, #0
	ldr r0, [r1] /* 读出原来的值备份 */
	str r1, [r1] /* 0->[0] */
	ldr r2, [r1] /* r2=[0] */
	cmp r1, r2   /* r1==r2? 如果相等表示是NAND启动 */
	ldr sp, =0x40000000+4096 /* 先假设是nor启动 */
	moveq sp, #4096  /* nand启动 */
	streq r0, [r1]   /* 恢复原来的值 */

	bl sdram_init
	//bl sdram_init2	 /* 用到有初始值的数组，不是位置无关码 */

	/* 重定位text, rodata, data段整个程序 */
	bl copy2sdram

	/* 清除BSS段 */
	bl clean_bss

	/* 复位之后，cpu处于svc模式
	 * 现在，切换到usr模式
	 */
	mrs r0, cpsr      /* 读出cpsr */
	bic r0, r0, #0xf  /* 修改M4-M0为0b10000，进入usr模式 */
```

```
    bic r0, r0, #(1<<7)  /* 清除 I 位, 使能中断 */
    msr cpsr, r0

    /* 设置 sp_usr */
    ldr sp, =0x33f00000

    ldr pc, =sdram
sdram:
    bl uart0_init

    bl print1
    /* 故意加入一条未定义指令 */
und_code:
    .word 0xdeadc0de  /* 未定义指令 */
    bl print2

    swi 0x123  /* 执行此命令, 触发 SWI 异常, 进入 0x8 执行 */

/***最后执行 main 函数***/
    //bl main  /* 使用 BL 命令相对跳转, 程序仍然在 NOR/sram 执
行 */
    ldr pc, =main  /* 绝对跳转, 跳到 SDRAM */

halt:
    b halt
```

进入 main.c 做一系列初始化

```
int main(void)
{
    led_init();
    //interrupt_init();  /* 初始化中断控制器 */
    key_eint_init();   /* 初始化按键, 设为中断源 */
    timer_init();
```

```
    puts("\n\rg_A = ");
    printHex(g_A);
    puts("\n\r");
```

进入按键初始化程序 interrupt.c

```
/* 初始化按键, 设为中断源 */

void key_eint_init(void)
{
    /* 配置 GPIO 为中断引脚 */
    GPFCON &= ~((3<<0) | (3<<4));
    GPFCON |= ((2<<0) | (2<<4));   /* S2,S3 被配置为中断引
脚 */

    GPGCON &= ~((3<<6) | (3<<22));
    GPGCON |= ((2<<6) | (2<<22));   /* S4,S5 被配置为中断引
脚 */


    /* 设置中断触发方式: 双边沿触发 */
    EXTINT0 |= (7<<0) | (7<<8);     /* S2,S3 */
    EXTINT1 |= (7<<12);            /* S4 */
    EXTINT2 |= (7<<12);            /* S5 */

    /* 设置 EINTMASK 使能 eint11,19 */
    EINTMASK &= ~((1<<11) | (1<<19));

/***注册中断控制器****/
    register_irq(0, key_eint_irq);
    register_irq(2, key_eint_irq);
    register_irq(5, key_eint_irq);
}
```

时钟初始化程序 timer_init();

```
void timer_init(void)
{
    /* 设置 TIMER0 的时钟 */
    /* Timer clk = PCLK / {prescaler value+1} /
{divider value}
                 = 50000000/(99+1)/16
                 = 31250
     */
    TCFG0 = 99;  /* Prescaler 0 = 99, 用于 timer0,1 */
    TCFG1 &= ~0xf;
    TCFG1 |= 3;  /* MUX0 : 1/16 */

    /* 设置 TIMER0 的初值 */
    TCNTB0 = 15625;  /* 0.5s 中断一次 */

    /* 加载初值, 启动 timer0 */
    TCON |= (1<<1);   /* Update from TCNTB0 & TCMPB0 */

    /* 设置为自动加载并启动 */
    TCON &= ~(1<<1);
    TCON |= (1<<0) | (1<<3);  /* bit0: start, bit3: auto
reload */

    /* 设置中断 */
    register_irq(10, timer_irq);
}
```

让后 main.c 函数一直循环执行 输出串口信息

```
while (1)
{
    putchar(g_Char);
    g_Char++;
```

```
    putchar(g_Char3);
    g_Char3++;
    delay(1000000);
    //printHex(TCNTO0);
}
```

定时器减到 0 的时候就会产生中断，start.S 跳到 0x18 的地方执行

```
    ldr pc, irq_addr /* vector 0x18 : irq */
    b halt              /* vector 0x1c : fiq */
.align 4

do_irq:
    /* 执行到这里之前:

     * 1. lr_irq 保存有被中断模式中的下一条即将执行的指令的地址

     * 2. SPSR_irq 保存有被中断模式的 CPSR

     * 3. CPSR 中的 M4-M0 被设置为 10010, 进入到 irq 模式

     * 4. 跳到 0x18 的地方执行程序
     */

    /* sp_irq 未设置, 先设置它 */
    ldr sp, =0x33d00000

    /* 保存现场 */

    /* 在 irq 异常处理函数中有可能会修改 r0-r12, 所以先保存 */

    /* lr-4 是异常处理完后的返回地址, 也要保存 */
    sub lr, lr, #4
    stmdb sp!, {r0-r12, lr}

    /* 处理 irq 异常 */
    bl handle_irq_c

    /* 恢复现场 */
```

```
    ldmia sp!, {r0-r12, pc}^  /* ^会把spsr_irq 的值恢复到
cpsr 里 */
```

看看怎么处理 irq

```
void handle_irq_c(void)
{
    /* 分辨中断源 */
    int bit = INTOFFSET;

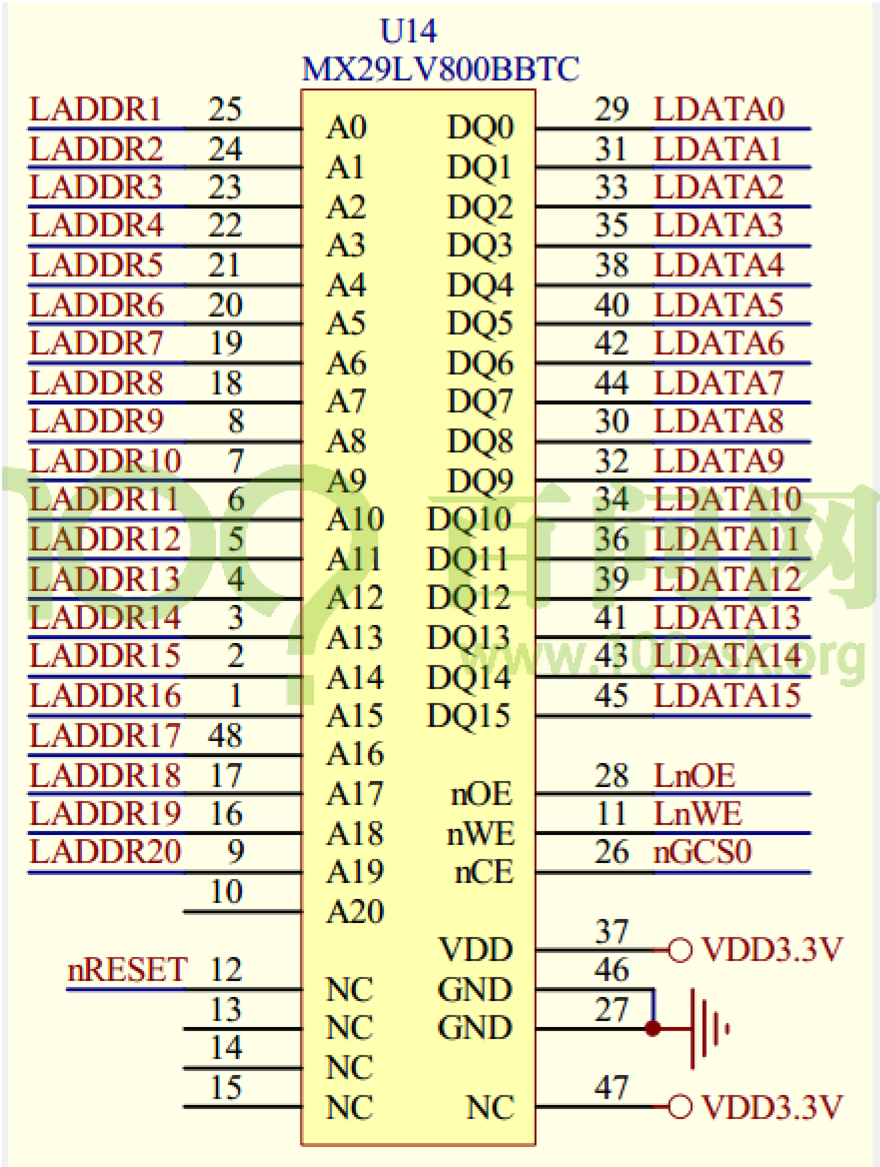
    /* 调用对应的处理函数执行 */
    irq_array[bit](bit);

    /* 清中断 : 从源头开始清 */
    SRCPND = (1<<bit);
    INTPND = (1<<bit);
}
```

第 015 课 NOR Flash

第 001 节_Nor Flash 原理及硬件操作

Nor Flash 的连接线有地址线，数据线，片选信号读写信号等，Nor Flash 的接口属于内存类接口，Nor Flash 可以向内存一样读，但是不能像内存一样写，需要做一些特殊的操作才能进行写操作，读只需像内存一样读很简单。Nor Flash 原理图如图：



Flash 介绍

常用的 Flash 类型有 Nor Flash 和 NAND Flash 两种。Nor Flash 由 Intel 公司在 1988 年发明，以替代当时在市场上占据主要地位的 EPROM 和 E2PROM。NAND Flash 由 Toshiba 公司在 1989 年发明。两者的主要差别如下表：

-	Nor	NAND
XIP(代码可以直接运行)	Yes	no
性能(擦除)	非常慢(5s)	快(3ms)
性能(写)	慢	快
性能(读)	快	快
可靠性	较高，位反转的比例小于 NAND Flash 的 10%	比较低，位反转比较常见，必须有校验措施，比如 TNR 必须有坏块管理措施

可擦除次数	10000 ~ 100000	100000 ~ 1000000
生命周期	低于 NAND Flash 的 10%	是 Nor Flash 的 10 倍以上
接口	与 RAM 接口相同	I/O 接口
易用性	容易	复杂
主要用途	常用于保存代码和关键数	用于保存数据
价格	高	低

Nor Flash 支持 XIP，即代码可以直接在 Nor Flash 上执行，无需复制到内存中。这是由于 NorF lash 的接口与 RAM 完全相同，可以随机访问任意地址的数据。Nor Flash 进行读操作的效率非常高，但是擦除和写操作的效率很低，另外，Nor Flash 的容量一般比较小。NAND Flash 进行擦除和写操作的效率更高，并且容量更大。一般而言，Nor Flash 用于存储程序，NAND Flash 用于存储数据。基于 NAND Flash 的设备通常也要搭配 Nor Flash 以存储程字。

Flash 存储器件由擦除单元（也称为块）组成，当要写某个块时，需要确保这个块已经 被擦除。Nor Flash 的块大小范围为 64kB、128kB：NAND Flash 的块大小范围为 8kB，64kB，擦/写一个 Nor Flash 块需 4s，而擦 / 写一个 NAND Flash 块仅需 2ms。Nor Flash 的块太大，不仅增加了擦写时间，对于给定的写操作，Nor Flash 也需要更多的擦除操作——特别是小文件，比如一个文件只有 IkB，但是为了保存它却需要擦除人小为 64kB—128kB 的 Nor Flash 块。

Nor Flash 的接口与 RAM 完全相同，可以随意访问任意地址的数据。而 NAND Flash 的 接口仅仅包含几个 I/O 引脚，需要串行地访问。NAND Flash 一般以 512 字节为单位进行读写。这使得 Nor Flash 适合于运行程序，而 NAND Flash 更适合于存储数据。

容量相同的情况下，NAND Flash 的体积更小，对于空间有严格要求的系统，NAND Flash 可以节省更多空间。市场上 Nor Flash 的容量通常为 IMB~4MB(也有 32MB 的 Nor Flash)，NAND Flash 的容量为 8MB~512MB。容量的差别也使得 Nor Flash 多用于存储程序，NAND Flash 多用于存储数据。

对于 Flash 存储器件的可靠性需要考虑 3 点：位反转、坏块和可擦除次数。所有 Flash 器件都遭遇位反转的问题：由于 Flash 固有的电器特性，在读写数据过程中，偶然会产生一位或几位数据错误（这种概率很低），而 NAND Flash 出现的概率远大于 Nor Flash，当位反转发生在关键的代码、数据上时，有可能导致系统崩溃。当仅仅是报告位反转，重新读取即可：如果确实发生了位反转，则必须有相应的错误检测/恢复措施。在 NAND Flash 上发生位反转的概率史高，推荐使用 EDC/ECC 进行错误检测和恢复。NAND Flash 上面会有坏块随机分布在使用前需要将坏块扫描出来，确保不再使用它们，否则会使产品含有严重的故障。NAND Flash 每块的可擦除次数通常在 100000 次左右，是 Nor Flash 的 10 倍。另外，因为 NAND Flash 的块大小通常是 NorF lash 的 1/8，所以 NAND Flash 的寿命远远超过 Nor Flash。

嵌入式 Linux 对 Nor、NAND Flash 的软件支持都很成熟。在 Nor Flash 上常用 jffs2 文 件系统，而在 NAND Flash 常用 yaffs 文件系统。在更底层，有 MTD 驱动程序实现对它们的读、写、擦除操作，它也实现了 EDC/ECC 校验。

Nor Flash 的操作

下面我们使用 u-boot 来体验 Nor Flash 的操作(开发板设置 Nor 启动，进入 u-boot)。 1)，使用 OpenJTAG 烧写 UBOOT 到 Nor Flash 那么我们怎么用 u-boot 来操作呢？ Nor Flash 手册里会有一个命令的表格，如图：

TABLE 5. MX29LV800BT/BB COMMAND DEFINITIONS

Command		Bus	First Bus Cycle		Second Bus Cycle		Third Bus Cycle		Fourth Bus Cycle		Fifth Bus Cycle		Sixth Bus Cycle	
		Cycle	Addr	Data	Addr	Data	Addr	Data	Addr	Data	Addr	Data	Addr	Data
Reset		1	XXXH	F0H										
Read		1	RA	RD										
Read Silicon ID	Word	4	555H	AAH	2AAH	55H	555H	90H	ADI	DDI				
	Byte	4	AAAH	AAH	555H	55H	AAAH	90H	ADI	DDI				
Sector Protect Verify	Word	4	555H	AAH	2AAH	55H	555H	90H	(SA)	XX00H				
									x02H	XX01H				
	Byte	4	AAAH	AAH	555H	55H	AAAH	90H	(SA)	00H				
									x04H	01H				
Program	Word	4	555H	AAH	2AAH	55H	555H	A0H	PA	PD				
	Byte	4	AAAH	AAH	555H	55H	AAAH	A0H	PA	PD				
Chip Erase	Word	6	555H	AAH	2AAH	55H	555H	80H	555H	AAH	2AAH	55H	555H	10H
	Byte	6	AAAH	AAH	555H	55H	AAAH	80H	AAAH	AAH	555H	55H	AAAH	10H
Sector Erase	Word	6	555H	AAH	2AAH	55H	555H	80H	555H	AAH	2AAH	55H	SA	30H
	Byte	6	AAAH	AAH	555H	55H	AAAH	80H	AAAH	AAH	555H	55H	SA	30H
Sector Erase Suspend		1	XXXH	B0H										
Sector Erase Resume		1	XXXH	30H										
CFI Query	Word	1	55H	98										
	Byte	1	AAH	98										

下面简单的举一些例子： 复位(reset)：往任何一个地址写入 F0。 读 ID(ReadSiliconID)：很多的 Nor Flash 可以配置成位宽 16bit(Word)，位宽 8bit(Byte)。对于我们使用的 jz2440 开发板使用是位宽 16bit，怎样读 ID 呢？根据前面得图可知，往 Nor Flash 的 555 地址写 AA，再往 2AA 的地址写入 55，再往 555 的地址写入 90，然后就可以读 ADI 地址，就可以读到 DDI 数据了。

实例 1

读数据： 在 u-boot 上执行：md.b0 结果(和我们烧进去的数据完全一样)：

00000000:170000ea14f09fe514f09fe514f09fe5................

00000010:14f09fe514f09fe514f09fe514f09fe5................

00000020:6001f833c001f8332002f8338002f833`..3...3..3...3

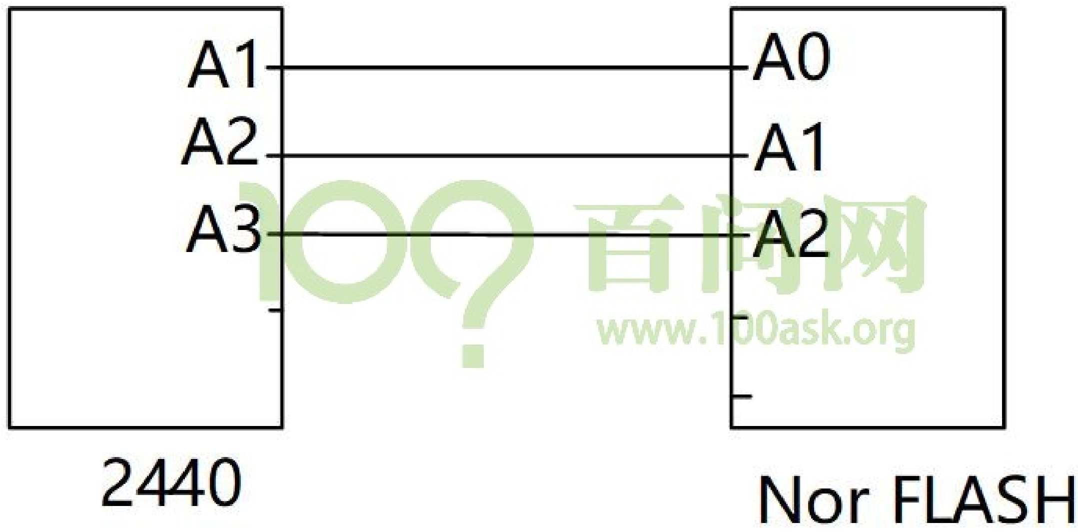
00000030:e002f8330004f8332004f833efbeadde...3...3..3....

可以得出结论：u-boot 可以像读内存一样来读 nor flash

实例 2

读 ID(参考 Nor 手册) 往地址 555H 写入 AAH(解锁) 往地址 2AAH 写入 55H(解锁) 往地址 555H 写入 90H（命令） 读 0 地址得到厂家 ID(C2H) 读 1 地址得到设备 ID(22DAH 或 225BH) 退出读 ID 状态：给任意地址写 F0H 就可以了。

下图为 2440 和 Nor Flash 的简易连接图：



2440 的 A1 接到 Nor 的 A0 所以 2440 发出的地址是，Nor Flash 收到的地址左移一位。比如：2440 发出(555H<<1)地址，Nor Flash 才能收到 555H 这个地址。下面对在 Nor Flash 的操作，2440 的操作，U-BOOT 上的操作进行比较，如下表：

Nor Flash 的操作	2440 的操作	U-BOOT 上的操作
往地址 555H 写入 AAH(解锁)	往地址 AAAH 写入 AAH(解锁)	mw.waaaaa
往地址 2AAH 写入 55H(解锁)	往地址 554H 写入 55H(解锁)	mw.w 554 55
往地址 555H 写入 90H（命令）	往地址 AAAH 写入 90H（命令）	mw.w aaa 90
读 0 地址得到厂家 ID(C2H)	读 0 地址得到厂家 ID(C2H)	md.w 0 1 (1:表示都一次)
读 1 地址得到设备 ID(22DAH 或 225BH)	读 2 地址得到设备 ID(22DAH 或 225BH)	md.w 2 1 (1:表示都一次)
退出读 ID 状态：给任意地址写 F0H 就可以了	退出读 ID 状态：给任意地址写 F0H 就可以了	mw.w 0 f0

1），当执行过 md.w 0 1 结果(输出厂家 ID)： 00000000:00c2..(00c2 就是厂家 ID) 2），当执行过 md.w 2 1 结果(输出设备 ID)： 00000002:2249I"(2249 就是设备 ID)

3），当执行 mw.w 0 f0，就退出读 ID 的状态， 执行：md.b0 结果：00000000:17.（读到的就是 Nor Flash 地址・0 的数据）

Nor Flash 的两种规范

通常内核里面要识别一个 Nor Flash 有两种方法： 一种是 jedec 探测，就是在内核里面事先定义一个数组，该数组里面放有不同厂家各个芯片的一些参

数，探测的时候将 flash 的 ID 和数组里面的 ID 一一比较，如果发现相同的，就使用该数组的参数。jedec 探测的优点就是简单，缺点是如果内核要支持的 flash 种类很多，这个数组就会很庞大。内核里面用 jedec 探测一个芯片时，是先通过发命令来获取 flash 的 ID，然后和数组比较，但是 flash.c 中连 ID 都是自己通过宏配置的。 一种是 CFI(commonoflashinterface)探测，就是直接发各种命令来读取芯片的信息，比如 ID、容量等，芯片本身就包含了电压有多大，容量有有多少等信息。 下面对在 Nor Flash 上操作，2440 上操作，U-BOOT 上操作 cfi 探测（读取芯片信息）进行比较参考芯片手册。

Nor Flash 上操作 cfi	2440 上操作 cfi	U-BOOT 上操作 cfi
往 55H 地址写入 98H	往 AAH 地址写入 98H	mw.w aa 98
读地址 10H 得到 0051	读地址 20H 得到 0051	md.w 20 1
读地址 11H 得到 0052	读地址 22H 得到 0052	md.w 22 1
读地址 12H 得到 0059	读地址 24H 得到 0059	md.w 24 1
读地址 27H 得到容量	读地址 4EH 得到容量	md.w 4e 1

Nor Flash 写数据

我们在 Nor Flash 的 10000 的地址读数据， md.w 100000 1 结果： 00100000:ffff.. 在 noroflash 的 10000 的地址写数据下 0x1234， mw.w 100000 1234 然后在这个地址读数据， md.w 100000 1 结果： 00100000:ffff(这个地址上的数据没有被修改，写操作无效)。

怎样把数据写进 Nor Flash 进去呢？参考图 1-3，写数据之前必须保证，要写的地址是擦除的。下面是 Nor Flash 的写操作，如下表：

Nor Flash 上操作写操作	2440 上操作写操作	U-BOOT 上操作写操作
往地址 555H 写 AAH(解锁)	往地址 AAAH 写 AAH(解锁)	mw.w aaa aa
往地址 2AAH 写 55H(解锁)	往地址 554H 写 55H(解锁)	mw.w 554 55
往地址 555H 写 A0H	往地址 AAAH 写 A0H	mw.w aaa a0
往地址 PA 写 PD	往地址 0x100000 写 1234h	mw.w 100000 1234

1)，U-BOOT 执行完上述指令后，0x1234，就被写到 0x100000 地址处，执行：md.w1000001 结果(1234 被写进去)： 00100000:1234 4 从这里可以看出来 U-BOOT 的操作不是很复杂。

2），我们再次往 0x100000 地址处，写入 0x5678,执行如下命令： mw.w aaa aa mw.w 554 55 mw.w aaa a0 mw.w 100000 5678 查看 0x100000 地址处的数据 md.w 100000 1 结果： 00100000:12300. 0x100000 地址处的数据不是 0x5678，写操作失败，失败的原因是，原来的数据已经是 0x1234 不是全 0xffff，再次写操作失败，(Nor Flash 只有先擦出，才能烧写)。

先擦除(参考 Nor Flash 芯片手册) Nor Flash 操作 u-boot 操作 555H AAH mw.w aaa aa 2AAH 55H mw.w 554 55 555H 80H mw.w aaa 80 555H AAH mw.w aaa aa 2AAH 55H mw.w 554 55 SA 30H //往扇区地址写入 30 mw.w 100000 30

执行完上述指令后测试 执行：md.w 100000 1 结果：00100000:ffff.. 已被擦除，这个时候再次烧写就不会有问题了。

再烧写 mw.w aaa aa mw.w 554 55 mw.w aaa a0 mw.w 100000 5678

测试烧写结果 执行：md.w 100000 1 结果：00100000:5678 xV 数据被烧写进去，烧写成功。

总结：我们烧写时，如果上面的数据，不是 0ffff,没有被擦除过，我们就要先擦出，擦除完后，才可以烧写，擦除烧写的命令可以从芯片手册里面获得。

第 002 节_Nor Flash 编程_识别

本节实例的目的目的：识别 nor flash 发送命令函数 nor_cmd 函数代码如下，往 NOR Flash 某个地址发送指令，

```
16
17  /* offset 是基于 NOR 的角度看到 */
18  void nor_cmd(unsigned int offset, unsigned int cmd)
19  {
20       nor_write_word(NOR_FLASH_BASE, offset, cmd);
21  }
```

读取函数

nor_read_word 函数是从 NOR Flash 读取两个字节（本开发板位宽 16bit），读取数据的地址，是基于 2440，所以读取 NOR Flash 某个地址上的数据时，需要把 NOR Flash 对应的地址左移一位(地址乘以 2)。

```
23  unsigned int nor_read_word(unsigned int base, unsigned int offset)
24  {
25       volatile unsigned short *p = (volatile unsigned short *)(base + (offset << 1));
26       return *p;
27  }
```

读取地址中的数据

向 nor_dat 函数中写入 NOR Flash 某个地址，返回该 NOR Flash 地址上的数据，

```
29  unsigned int nor_dat(unsigned int offset)
30  {
31       return nor_read_word(NOR_FLASH_BASE, offset);
32  }
```

进入 NOR FLASH 的 CFI 模式，读取各类信息 do_scan_nor_flash 函数代码如下，该函数的功能：进入 CFI 模式读取 NOR Flash 中的厂家 ID，设备 ID,容量等信息。

```
50/*  进入 NOR FLASH 的 CFI 模式
51 *  读取各类信息
52 */
53  void do_scan_nor_flash(void)
54  {
55        char str[4];
56        unsigned int size;
57        int regions, i;
58        int region_info_base;
59        int block_addr, blocks, block_size, j;
60        int cnt;
61
62        int vendor, device;
63
64        /*  打印厂家 ID、设备 ID */
65        nor_cmd(0x555, 0xaa);        /*  解锁  */
66        nor_cmd(0x2aa, 0x55);
67        nor_cmd(0x555, 0x90);        /* read id */
68        vendor = nor_dat(0);
69        device = nor_dat(1);
70        nor_cmd(0, 0xf0);            /* reset */
71
72        nor_cmd(0x55, 0x98);    /*  进入 cfi 模式  */
073
74        str[0] = nor_dat(0x10);
75        str[1] = nor_dat(0x11);
76        str[2] = nor_dat(0x12);
77        str[3] = '\0';
78        printf("str = %s", str);
79
80        /*  打印容量  */
81        size = 1<<(nor_dat(0x27));
82        printf("vendor id = 0x%x, device id = 0x%x, nor size = 0x%x, %dM", 
vendor, device, size, size/(1024*1024));
83
84        /*  打印各个扇区的起始地址  */
85        /*  名词解释:
86         *      erase block region :  里面含有 1 个或多个 block,  它们的大小
一样
87         *  一个 nor flash 含有 1 个或多个 region
88         *  一个 region 含有 1 个或多个 block(扇区)
```

89
90 * Erase block region information:
91 * 前 2 字节+1 : 表示该 region 有多少个 block
92 * 后 2 字节*256 : 表示 block 的大小
93 */
94
95 regions = nor_dat(0x2c);
96 region_info_base = 0x2d;
97 block_addr = 0;
98 printf("Block/Sector start Address:");
99 cnt = 0;
100 for (i = 0; i < regions; i++)
101 {
102 blocks = 1 + nor_dat(region_info_base) + (nor_dat(region_info_base+1)<<8);
103 block_size = 256 * (nor_dat(region_info_base+2) + (nor_dat(region_info_base+3)<<8));
104 region_info_base += 4;
105
106 //printf("region %d, blocks = %d, block_size = 0x%x, block_addr = 0x%x", i, blocks, block_size, block_addr);
107
108 for (j = 0; j < blocks; j++)
109 {
110 /* 打印每个 block 的起始地址 */
111 //printf("0x%08x ", block_addr);
112 printHex(block_addr);
113 putchar(' ');
114 cnt++;
115 block_addr += block_size;
116 if (cnt % 5 == 0)
117 printf("\n\r");
118 }
119 }
120 printf("\n\r");
121 /* 退出 CFI 模式 */
122 nor_cmd(0, 0xf0);
123 }

第 65，66 行 这两步是解锁，解锁之后就进入读 ID 状态，就可以读取厂家和设备 ID 了。

第 68 行 是把读取到的厂家 ID 的值，复制给 vendor 变量。

第 69 行 是把读取到的设备 ID 的值，复制给 device 变量。

第 70 行 退出读 ID 状态: 给任意地址写 F0H。

第 72 行,往地址 0x55 地址写入数据 0x98,是进入 cfi 模式。

第 74，75，76 行是读取 NOR Flash 地址 0x10,0x11,x012 中的字符，赋值给字符串 str。

第 81 行，根据芯片手册可知道，读取 NOR Flash 地址 0x27 处的数据，得到的是 NOR Flash 容量大小 2 的幂数，所以把 1 左移读取到的数据，就可得到 NOR Flash 的容量。

第 95 行读取 NOR Flash 地址 0x2c 地址中的数据，可以得到 NOR Flash 中有多少 region。

第 102 行根据 Erase block region information:的信息可以知道读取[2E,2D]这两个字节的地址+1，可以得到一个 region 有多少 block(参考芯片手册)。代码中的 region_info_base 变量的值是 0x2d,0x2d 是前两个字节中的低字节，0x2e 是前两个字节中的高字节，所以需要左移 8 位，然后加上 1 就得到了一个 region 有多少 block.。

第 103 行参考芯片手册，读取[30,2F]这两个字节地址，然后乘上 256 就可以得到一个块的大小。

第 104 行，地址加 4，读取下一个 region 有多少 block 和每个 block 的大小。

第 112,115 行，由于 NOR Flash 的基地址是 0，所以第一个 block 的首地址是 0，下一个 block 的首地址，就是上一个 block 的首地址加上 block 的大小。

第 112 行往 0 地址写入 0xf0,退出 CFI 模式。

Nor Flash 的测试 nor_flash_test 函数通过 switch 语句，分别处理识别 NOR Flash，擦除 NOR Flash 某个扇区，编写某个地址，读某个地址。代码如下：

```
232    void nor_flash_test(void)
233    {
234        char c;
235
236        while (1)
237        {
238            /*  打印菜单, 供我们选择测试内容  */
239            printf("[s] Scan nor flash\n\r");
240            printf("[e] Erase nor flash\n\r");
241            printf("[w] Write nor flash\n\r");
242            printf("[r] Read nor flash\n\r");
243            printf("[q] quit\n\r");
244            printf("Enter selection: ");
245
246            c = getchar();
247            printf("%c", c);
248
249            /*  测试内容:
250             * 1.  识别 nor flash
251             * 2.  擦除 nor flash 某个扇区
252             * 3.  编写某个地址
253             * 4.  读某个地址
```

```
254                  */
255                 switch (c)
256                 {
257                     case 'q':
258                     case 'Q':
259                         return;
260                         break;
261
262                     case 's':
263                     case 'S':
264                         do_scan_nor_flash();
265                         break;
266
267                     case 'e':
268                     case 'E':
269                         do_erase_nor_flash();
270                         break;
271
272                     case 'w':
273                     case 'W':
274                         do_write_nor_flash();
275                         break;
276
277                     case 'r':
278                     case 'R':
279                         do_read_nor_flash();
280                         break;
281                     default:
282                         break;
283                 }
284             }
285     }
```

主函数 main 函数代码如下所示。把 timer 中断去掉，否则：测试 NOR Flash 时进入 CFI 等模式时，如果发生了中断，cpu 必定读 NOR Flash，那么读不到正确的指令，导致程序崩溃。

```
12  int main(void)
13  {
14      led_init();
15      //interrupt_init();   /*  初始化中断控制器  */
16      key_eint_init();      /*  初始化按键，设为中断源  */
17      //timer_init();
18
19      puts("\n\rg_A = ");
20      printHex(g_A);
```

```
21        puts("\n\r");
22
23        nor_flash_test();
24
25        return 0;
26  }
```

第 003 节_Nor Flash 编程_擦写读

本实例的目的目的：擦除 nor flash 某个扇区，编写某个地址，读某个地址。

等待烧写 等待烧写完成：读数据, Q6 无变化时表示结束 (参考芯片手册)，

```
35  void wait_ready(unsigned int addr)
36  {
37        unsigned int val;
38        unsigned int pre;
39
40        pre = nor_dat(addr>>1);
41        val = nor_dat(addr>>1);
42        while ((val & (1<<6)) != (pre & (1<<6)))
43        {
44              pre = val;
45              val = nor_dat(addr>>1);
46        }
47}
```

擦除 NOR Flash 某个扇区 do_erase_nor_flash 函数的代码如下。参考芯片手册，就可以知道擦除某个扇区，还是相对比较简单的。

```
125       void do_erase_nor_flash(void)
126       {
127             unsigned int addr;
128
129                   /* 获得地址 */
130             printf("Enter the address of sector to erase: ");
131             addr = get_uint();
132
133             printf("erasing ...");
134             nor_cmd(0x555, 0xaa);       /* 解锁 */
135             nor_cmd(0x2aa, 0x55);
136             nor_cmd(0x555, 0x80);      /* erase sector */
137
138             nor_cmd(0x555, 0xaa);       /* 解锁 */
139             nor_cmd(0x2aa, 0x55);
140             nor_cmd(addr>>1, 0x30);  /* 发出扇区地址 */
```

```
141         wait_ready(addr);
142     }
```

第 131 行，get_uint 函数用于获取输入的地址。

第 134,135 这两行是解锁。

第 136 行是 erase sector。

第 138,139 行是再次解锁。

第 140 行是对发出的扇区地址。

第 141 行等待擦除完成。

写 NOR Flash do_write_nor_flash 的代码如下所示，开发板上的 NOR Flash 的位宽是 16bit,所以可以把要写的数据构造出 16bit 然后在写进 NOR Flash 中。

```
144     void do_write_nor_flash(void)
145     {
146         unsigned int addr;
147         unsigned char str[100];
148         int i, j;
149             unsigned int val;
150
151         /*  获得地址  */
152         printf("Enter the address of sector to write: ");
153         addr = get_uint();
154
155         printf("Enter the string to write: ");
156         gets(str);
157
158         printf("writing ...\n\r");
159
160         /* str[0],str[1]==>16bit
161          * str[2],str[3]==>16bit
162         */
163         i = 0;
164         j = 1;
165         while (str[i] && str[j])
166         {
167             val = str[i] + (str[j]<<8);
168
169             /*  烧写  */
170             nor_cmd(0x555, 0xaa);      /*  解锁  */
171             nor_cmd(0x2aa, 0x55);
172             nor_cmd(0x555, 0xa0);      /* program */
173             nor_cmd(addr>>1, val);
174             /*  等待烧写完成 ：读数据, Q6 无变化时表示结束  */
175             wait_ready(addr);
176
177             i += 2;
```

```
178             j += 2;
179             addr += 2;
180         }
181 
182         val = str[i];
183         /*  烧写  */
184         nor_cmd(0x555, 0xaa);      /*  解锁  */
185         nor_cmd(0x2aa, 0x55);
186         nor_cmd(0x555, 0xa0);      /* program */
187         nor_cmd(addr>>1, val);
188         /*  等待烧写完成 ：读数据, Q6 无变化时表示结束  */
189         wait_ready(addr);
190     }
```

第 153 行把通过 get_uint 获得的地址赋值给 addr 变量，

第 156 行通过 gets 函数获得输入的字符串。

第 168 行两个 8 位的数据，组合成一个 16 位的数据赋值给变量 val。

读 NOR Flash do_read_nor_flash 函数代码如下，由于 NOR Flash 是内存类接口，可以像内存一样读取。

```
191     void do_read_nor_flash(void)
192     {
193         unsigned int addr;
194         volatile unsigned char *p;
195         int i, j;
196         unsigned char c;
197         unsigned char str[16];
198 
199         /*  获得地址  */
200         printf("Enter the address to read: ");
201         addr = get_uint();
202 
203         p = (volatile unsigned char *)addr;
204 
205         printf("Data :    \n\r");
206         /*  长度固定为 64 */
207         for (i = 0; i < 4; i++)
208         {
209             /*  每行打印 16 个数据  */
210             for (j = 0; j < 16; j++)
211         {
212                 /*  先打印数值  */
213                 c = *p++;
214                 str[j] = c;
215                 printf("%02x ", c);
```

```
216             }
217
218             printf("      ; ");
219
220             for (j = 0; j < 16; j++)
221             {
222                 /*  后打印字符  */
223                 if (str[j] < 0x20 || str[j] > 0x7e)    /*  不可视字符  */
224                     putchar('.');
225                 else
226                     putchar(str[j]);
227             }
228             printf("\n\r");
229     }
```

第 201 行中的 get_uint 函数，从串口中获得输入的地址。

第 203 行，强制类型转化。

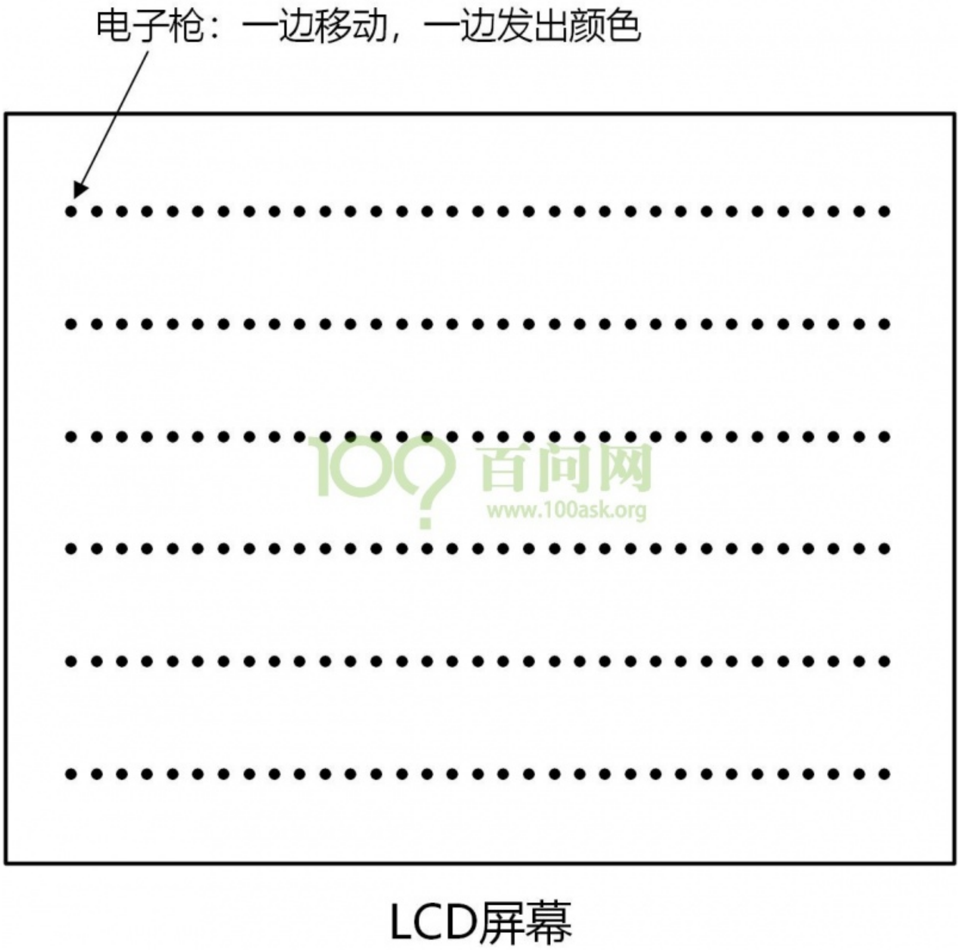
第 207 行~216 行是对 NOR Flash 内容的读取，输出的内容为 16 进制的数据，由于 NOR Flash 是内存类接口，可以像内存一样读取。

第 220 行~227 输出 NOR Flash 的内容为字符型数据，其中的第 223 行用来判断，输出的字符是否为不可视字符，要是为不可视字符输出点'.',要是可视字符输出字符。

第 017 课 LCD

第 001 节_LCD 硬件原理

先简单介绍下 LCD 的操作原理。 如下图的 LCD 示意图，里面的每个点就是一个像素



点。

想象有一个电子枪，一边移动，一边发出各种颜色的光。这里有很多细节问题，我们一个一个的梳理。 1. 电子枪是如何移动的？ 答：有一条 CLK 时钟线与 LCD 相连，每发出一次 CLK(高低电平)，电子枪就移动一个像素。

2. 颜色如何确定？ 答：由连接 LCD 的三组线：R(Red)、G(Green)、B(Blue)确定。

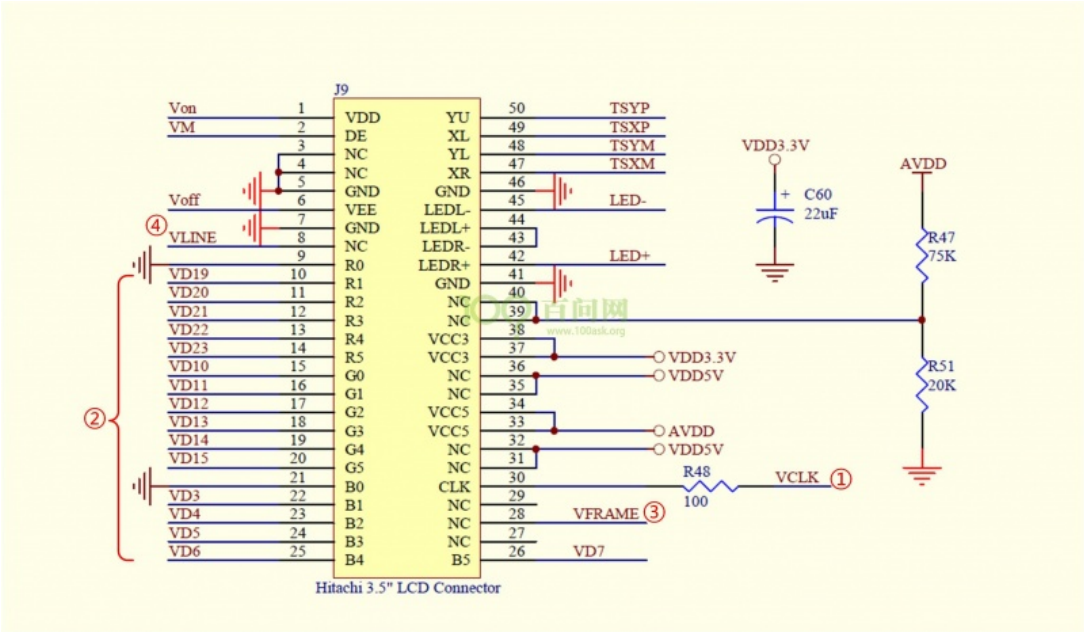
3. 电子枪如何得知应跳到下一行？ 答：有一条 HSYNC 信号线与 LCD 相连，每发出一次脉冲(高低电平)，电子枪就跳到下一行。

4. 电子枪如何得知应跳到原点？ 答：有一条 VSYNC 信号线与 LCD 相连，每发出一次脉冲(高低电平)，电子枪就跳到原点。

5. RGB 线上的数据从何而来？ 答：内存里面划分一块显存(FrameBuffer)，里面存放了要显示的数据，LCD 控制器从里面将数据读出来，通过 RGB 三组线传给电子枪，电子枪再依次打到显示屏上。

6. 前面的信号由谁发给 LCD？ 答：有 S3C2440 里面的 LCD 控制器来控制发出信号。

通过 JZ2440 原理图对上面进行验证，下图的 LCD 控制器接口图。



①是时钟信号，每来一个 CLK，电子枪就移动一个像素； ②是用来传输颜色数据； ③是垂直方向同步信号，FRAME(帧)； ④是水平方向同步信号，LINE(行)；

再来看看 LCD 的芯片手册。

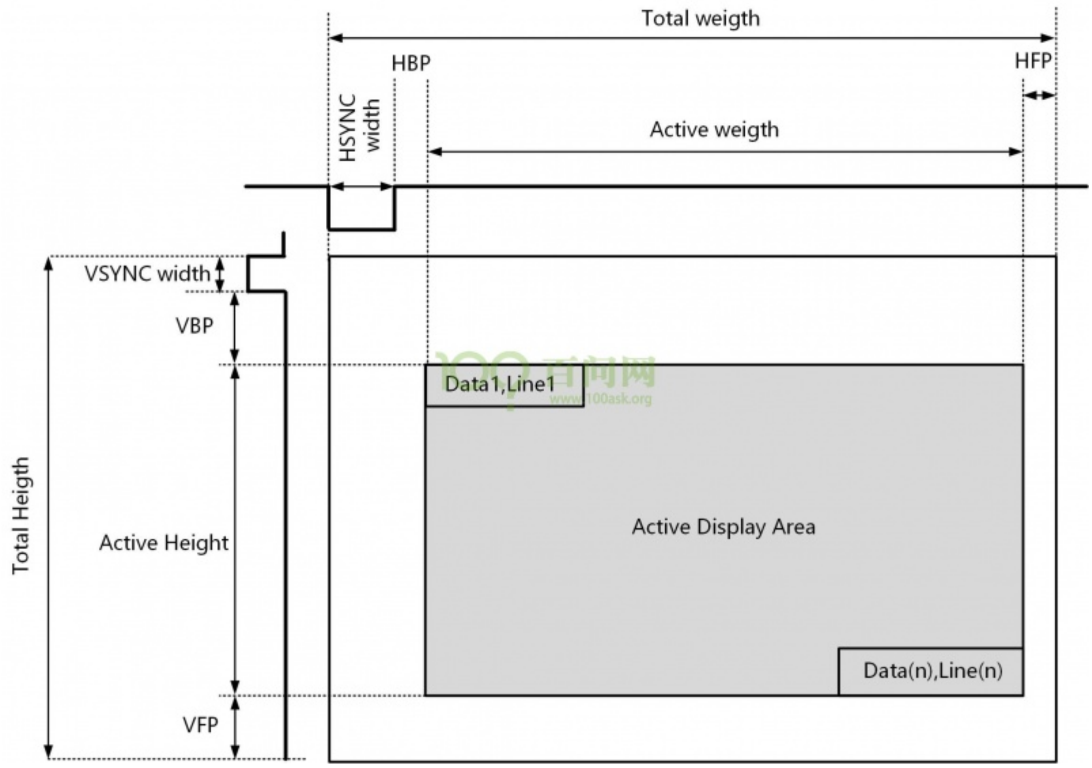
Pin No.	Symbol	I/O	Function	Remark
1	V_{LED-}	P	Power for LED backlight cathode	
2	V_{LED+}	P	Power for LED backlight anode	
3	GND	P	Power ground	
4	V_{DD}	P	Power voltage	
5	R0	I	Red data (LSB)	
6	R1	I	Red data	
7	R2	I	Red data	
8	R3	I	Red data	
9	R4	I	Red data	
10	R5	I	Red data	
11	R6	I	Red data	
12	R7	I	Red data (MSB)	
13	G0	I	Green data (LSB)	
14	G1	I	Green data	
15	G2	I	Green data	
16	G3	I	Green data	
17	G4	I	Green data	
18	G5	I	Green data	
19	G6	I	Green data	
20	G7	I	Green data (MSB)	
21	B0	I	Blue data (LSB)	
22	B1	I	Blue data	
23	B2	I	Blue data	
24	B3	I	Blue data	
25	B4	I	Blue data	
26	B5	I	Blue data	
27	B6	I	Blue data	
28	B7	I	Blue data (MSB)	
29	GND	P	Power ground	
30	PCLK	I	Pixel clock	
31	DISP	I	Display on/off	

先是 VLED+、VLED-背光灯电源。VDD、VDD 是 LCD 电源。 R0-R7、G0-G7、B0-B7 是红绿蓝颜色信号。 PCLK 是像素时钟信号。DISP 是像素开关。HSYNC、VSYNC 分别是水平方向、垂直方向信号。 DE 数据使能。X1、Y1、X2、Y2 是触摸屏信号。

可以看出 LCD 有很多信号，这些信号要根据时序图传输才能正确显示。参考 JZ2440_4.3 寸 LCD 手册_AT043TN24 的时序如下：

生成缩略图出错：尺寸超过 12.5 MP 的文件

从最小的像素开始分析，电子枪每次在 CLK 下降沿(本开发板是下降沿)从数据线 Dn0-Dn7 上得到数据，发射到显示屏上，然后移动到下一个位置。Dn0-Dn7 上的数据来源就是前面介绍的 FrameBuffer。就这样从一行的最左边，一直移动到一行的最右边，完成了一行的显示，假设为 x。 当打完一行的最后一个数据后，就会收到 Hsync 行同步信号，根据时序图，一个 Hsync 周期可以大致分为五部分组成：thp、thb、1/tc、thd、thf。thp 称为脉冲宽度，这个时间不能太短，太短电子枪可能识别不到。电子枪正确识别到 thp 后，会从最右端移动最左端，这个移动的时间就是 thb，称之为移动时间。thf 表示显示完最右像素，再过多久 Hsync 才来。 同理，当电子枪一行一行的从上面移动到最下面时，Vsync 垂直同步信号就让电子枪移动回最上边。Vsync 中的 tvp 是脉冲宽度，tvb 是移动时间，tvf 表示显示完最下一行像素，再过多久 Vsync 才来。 假设一共有 y 行，则 LCD 的分辨率就是 x*y。 关于显示原理，可以参考这篇博客：http://www.cnblogs.com/shangdawei/p/4760933.html 里面有一个 LCD 显示配置示意图如
下：



当发出一个 HSYNC 信号后，电子枪就会从最右边花费 HBP 时长移动到最左边，等到了最右边后，等待 HFP 时长 HSYNC 信号才回来。因此，HBP 和 HFP 分别决定了左边和右边的黑框。 同理，当发出一个 VSYNC 信号后，电子枪就

会从最下边花费 VBP 时长移动到最上边，等到了最下边后，等待 VFP 时长 VSYNC 信号才回来。因此，VBP 和 VFP 分别决定了上边和下边的黑框。 中间灰色区域才是有效显示区域。

再来解决最后一个问题：每个像素再 FrameBuffer 中，占据多少位 BPP(Bits Per Pixels)？ 前面的 LCD 引脚功能图里，R0-R7、G0-G7、B0-B7，每个像素是占据 3*8=24 位的，即硬件上 LCD 的 BPP 是确定的。 虽然 LCD 上的引脚是固定的，但我们使用的时候，可以根据实际情况进行取舍，比如我们的 JZ2440 使用的是 16BPP，因此 LCD 只需要 R0-R4、G0-G5、B0-B4 与 SOC 相连，5+6+6=16BPP，每个像素就只占据 16 位数据。

我们写程序的思路如下： 1. 查看 LCD 芯片手册，查看相关的时间参数、分辨率、引脚极性； 2. 根据以上信息设置 LCD 控制器寄存器，让其发出正确信号； 3. 在内存里面分配一个 FrameBuffer，在里面用若干位表示一个像素，再把首地址告诉 LCD 控制器；

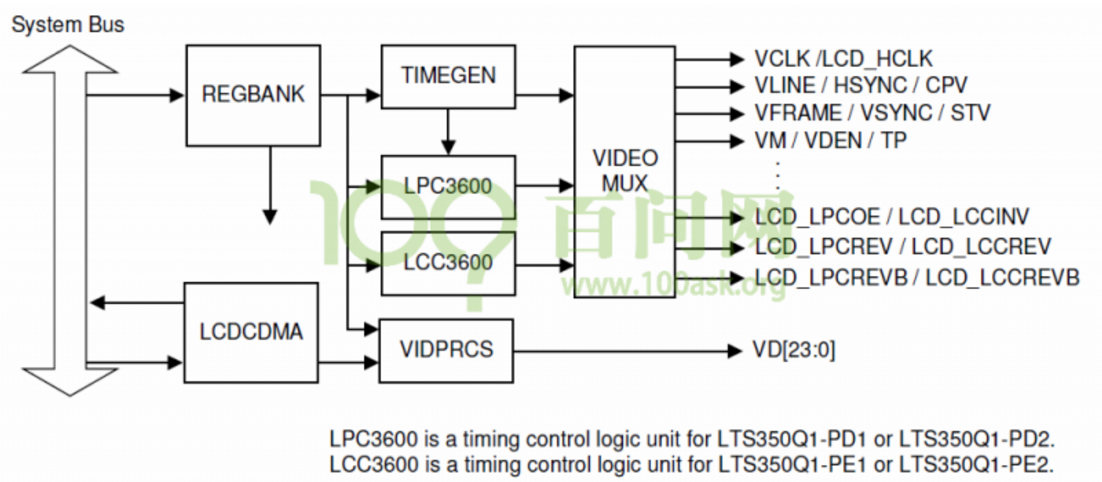
之后 LCD 控制器就能周而复始取出 FrameBuffer 里面的像素数据，配合其它控制信号，发送给电子枪，电子枪再让在 LCD 上显示出来。以后我们想显示图像，只需要编写程序向 FrameBuffer 填入相应数据即可，硬件会自动的完成显示操作。

第 002 节_S3C2440_LCD 控制器

LCD 控制器主要功能和需要的设置： 1. 取：从内存(FrameBuffer)取出某个像素的数据；之后需要把 FrameBuffer 地址、BPP、分辨率告诉 LCD 控制器； 2. 发：配合其它信号把 FrameBuffer 数据发给 LCD；需要设置 LCD 控制器时序、设置引脚极性；

这里主要的难点就是如何配合其它信号，需要我们阅读 LCD 芯片手册，知道其时序要求，然后设置相应的 LCD 控制器。

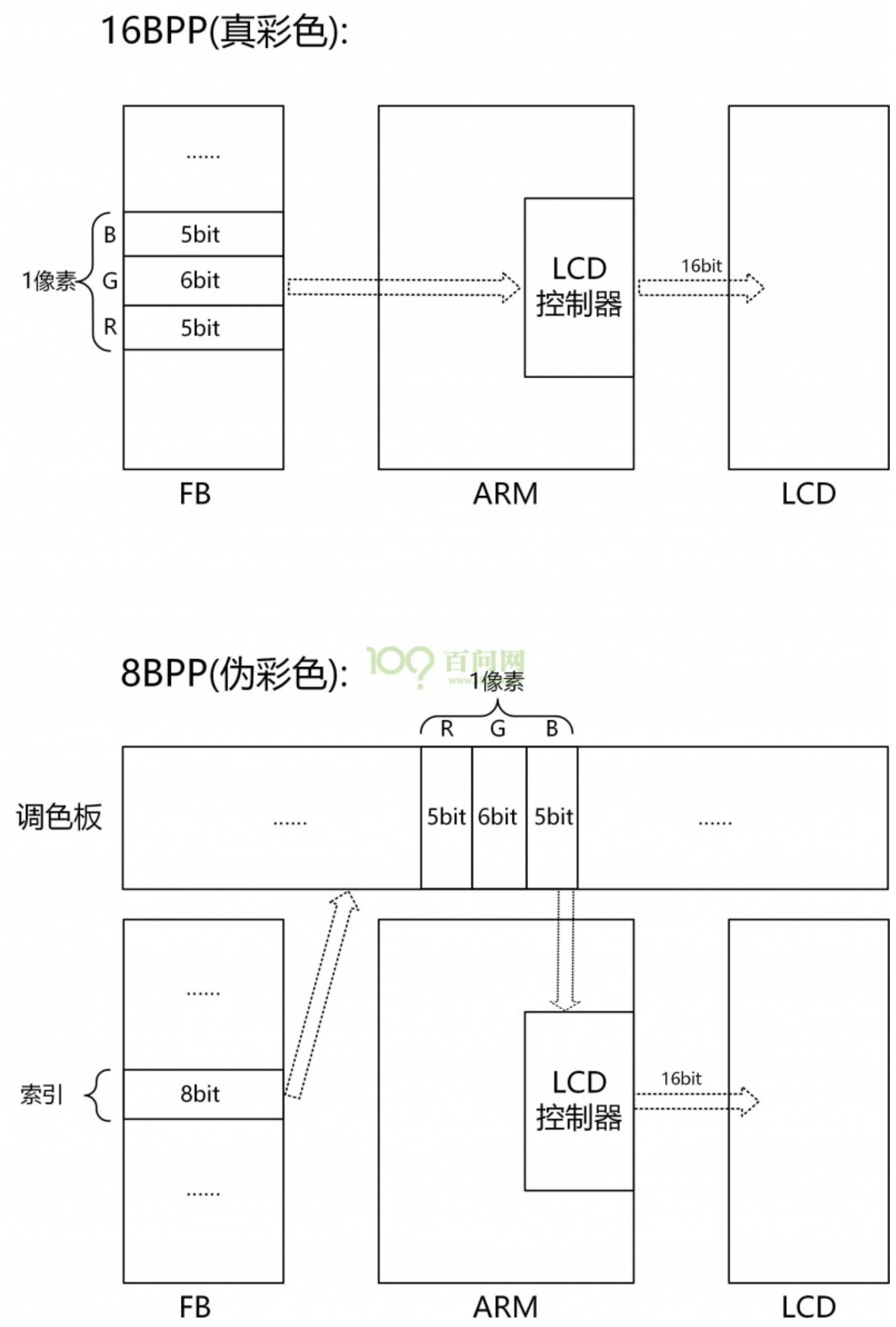
先看下 S3C2440 芯片手册上的 LCD 控制器框图：



通过设置 REGBANK(寄存器组)，LCDCDMA 会自动(无需 CPU 参与)把内存上 FrameBuffer 里的数据，通过 VIDPRCS 发送到引脚 VD[23:0]上，再配合 VIDEOMUX 引脚的控制信号，正确的显示出来。

S3C2440 芯片手册介绍了 LCD 控制器支持 TFT 和 STN 两种 LCD，我们常用的都是 TFT 材质的，因此主要看 TFT 相关的部分。

调色板的概念：　画油画的时候，通常先在调色板里配好想要的颜色，再用画笔沾到画布上作画。LCD 控制器里也借用了这个概念，从 FrameBuffer 获得数据，这个数据作为索引从调色板获得对应数据，再发给电子枪显示出来。



如图，假如是 16BPP 的数据，LCD 控制器从 FB 取出 16bit 数据，显示到 LCD

上。 当如果想节约内存，对颜色要求也没那么高，就可以采用调色板的方式，调色板里存放了 256 个 16bit 的数据，FB 只存放每个像素的索引，根据索引去调色板找到对应的数据传给 LCD 控制器，再通过电子枪显示出来。

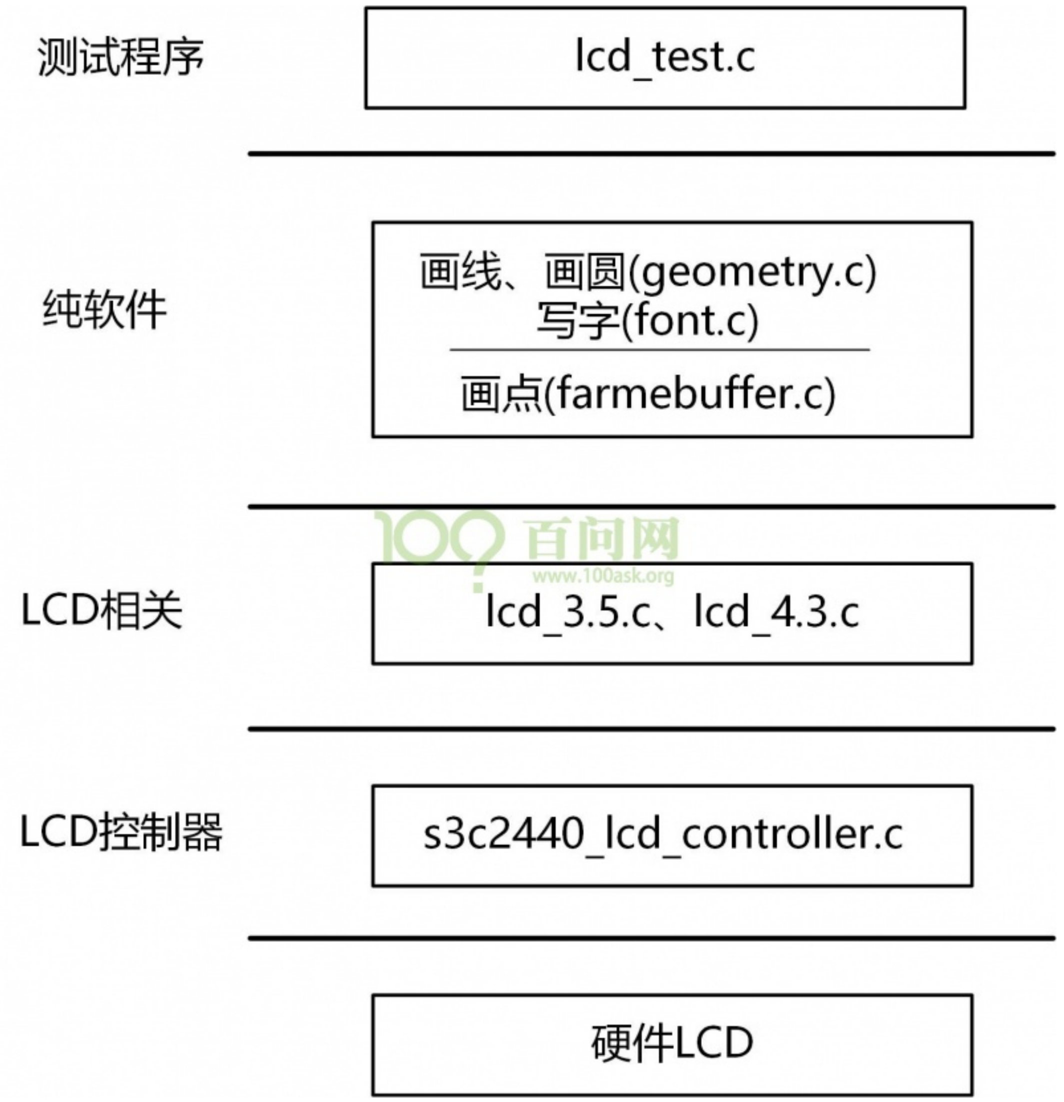
假设现在想要 LCD 只显示一种颜色怎么办？ 如果是 16BPP/24BPP 需要修改 FB 里面的数据，填充同一个值。 如果是 8BPP 可以修改 FB 为同一种颜色，也可以设置调色板为同一种颜色，对于 S3C22440 有个临时调色板的特性，一旦使能了临时调色板，不管 FB 里面是什么数据，都只调用临时调色板的数据。

第 003 节_编程_框架与准备

本节主要有两个目的： a. 讲解后续程序的框架； b. 准备一个支持 NAND、NOR 启动的程序；

我们的目的是在 LCD 显示屏上画线、画圆(geomentry.c)和写字(font.c)其核心是画点(farmebuffer.c)，这些都属于纯软件。此外还需要一个 lcd_test.c 测试程序提供操作菜单，调用画线、画圆和写字操作。 往下操作的是 LCD 相关的内容，不同的 LCD，其配置的参数也会不一样，通过 lcd_3.5.c 或 lcd_4.3.c 来设置。 根据 LCD 的特性，来设置 LCD 控制器，对于我们开发板，就是 s3c2440_lcd_controller.c，假如希望在其它开发板上也实现 LCD 显示，只需添加相应的代码文件即可。 这就是 LCD 编程的框架，尽可能的“高内聚低耦

合”。

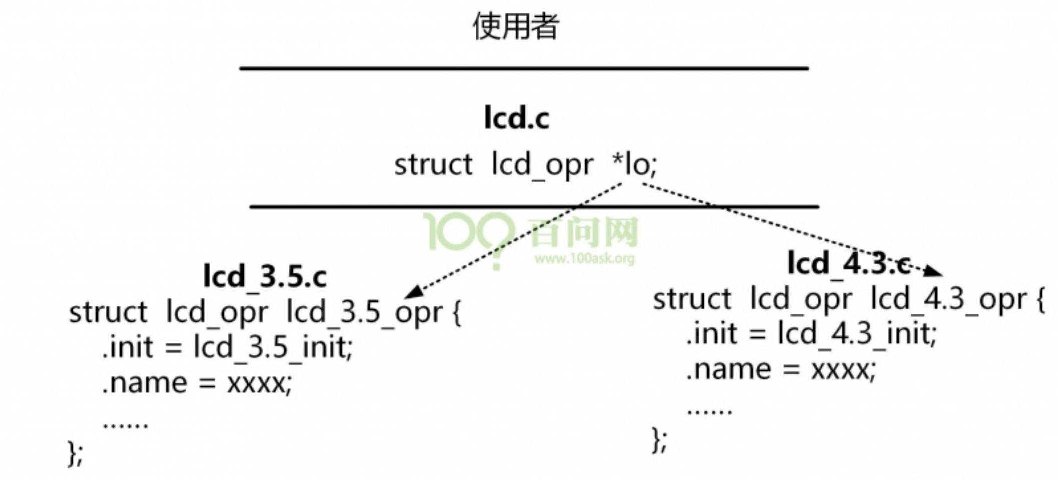


为了让程序更加好扩展，下面介绍“面向对象编程”的概念。 假如我们写好程序后，有两款尺寸大小的 lcd，如何快速的在两个 lcd 上切换？ 首先我们抽象出 lcd_3.5.c 和 lcd_4.3.c 的共同点，比如都有初始化函数 init(),我们可以新建一个 lcd.c，然后定义一个结构体：

```
struct lcd_opr{
        void (*init)(void);
};
```

用户不接触 lcd_3.5.c 和 lcd_4.3.c，只需要在 lcd.c 里通过指针访问对应的结构体的函数，也就调用了不同

init()。



前面我们的程序大小都没超过 4K，因此无论 Nor/Nand 启动，都是正常的，现在的 LCD 相关代码比较大，超过 4K，因此需要修改启动部分的代码。目前还未讲解 nand flash，因此直接将 19 课准备的 nand_flash 程序部分复制到当前代码里即可，关于这部分可以参考 nand flash 讲解部分。

第 004 节_编程_抽象出重要结构体

开始正式编写程序，根据前面的框架，新建如下文件： font.c、framebuffer.c、geometry.c、lcd.c、lcd_4.3.c、lcd_controller.c、s3c2440_lcd_controller.c、lcd_test.c

首先编写`lcd_controller.c`，它向上要接收不同 LCD 的参数，向下要使用这些参数设置对应的 LCD 控制器。 前面我们列举了 LCD 的参数，例如引脚的极性、时序、数据的格式 bpp、分辨率等，使用面向对象的思维方式，将这些封装成结构体放在`lcd.h`中：

```
enum {
    NORMAL = 0,
    INVERT = 1,
};

/* NORMAL : 正常极性
 * INVERT : 反转极性
 */
typedef struct pins_polarity {
    int vclk;   /* normal: 在下降沿获取数据 */
    int rgb;    /* normal: 高电平表示 1 */
    int hsync; /* normal: 高脉冲 */
    int vsync; /* normal: 高脉冲 */
}pins_polarity, *p_pins_polarity;

typedef struct time_sequence {
    /* 垂直方向 */
    int tvp; /* vysnc 脉冲宽度 */
```

```
    int tvb; /*  上边黑框, Vertical Back porch */
    int tvf; /*  下边黑框, Vertical Front porch */

    /*  水平方向  */
    int thp; /* hsync 脉冲宽度  */
    int thb; /*  左边黑框, Horizontal Back porch */
    int thf; /*  右边黑框, Horizontal Front porch */

    int vclk;
}time_sequence, *p_time_sequence;


typedef struct lcd_params {
    /*  引脚极性  */
    pins_polarity pins_pol;

    /*  时序  */
    time_sequence time_seq;

    /*  分辨率, bpp */
    int xres;
    int yres;
    int bpp;

    /* framebuffer 的地址  */
    unsigned int fb_base;
}lcd_params, *p_lcd_params;
```

以后就使用`lcd_params`结构体来表示 lcd 参数。

对于有多个 lcd 的情况，再定义个一个结构体，包含指针初始化函数和使能函数，放在`lcd_controller.h`里面：

```
typedef struct lcd_controller {
    void (*init)(p_lcd_params plcdparams);
    void (*enable)(void);
    void (*disable)(void);
}lcd_controller, *p_lcd_controller;
```

最后在`lcd_controller.c`里传入 lcd 参数，再通过指针函数初始化对应的 lcd 控制器：

```
void lcd_controller_init(p_lcd_params plcdparams)
{
    /*  调用 2440 的 LCD 控制器的初始化函数  */
    lcd_controller.init(plcdparams);
}
```

在`s3c2440_lcd_controller.c`还需构造一个当前 soc 的 lcd 控制器结构体：

```
struct lcd_controller s3c2440_lcd_controller = {
```

```
    .init      = xxx,
    .enalbe    = xxx,
    .disable = xxx,
};
```

第 005 节_编程_LCD 控制器

继续上一节的代码，修改`s3c2440_lcd_controller.c`：

```
struct lcd_controller s3c2440_lcd_controller = {
    .init      = s3c2440_lcd_controller_init,
    .enalbe    = s3c2440_lcd_controller_enalbe,
    .disable = xs3c2440_lcd_controller_disable,
};
```

然后对每个函数进行功能实现，首先是`s3c2440_lcd_controller_init`，依次设置 LCD 控制器寄存器，先是 LCD 寄存器 1：

LCD Control 1 Register

Register	Address	R/W	Description	Reset Value
LCDCON1	0X4D000000	R/W	LCD control 1 register	0x00000000

LCDCON1	Bit	Description	Initial State
LINECNT (read only)	[27:18]	Provide the status of the line counter. Down count from LINEVAL to 0	0000000000
CLKVAL	[17:8]	Determine the rates of VCLK and CLKVAL[9:0]. **STN**: VCLK = HCLK / (CLKVAL x 2) (CLKVAL ≥2) **TFT**: VCLK = HCLK / [(CLKVAL+1) x 2] (CLKVAL ≥ 0)	0000000000
MMODE	[7]	Determine the toggle rate of the VM. 0 = Each Frame 1 = The rate defined by the MVAL	0
PNRMODE	[6:5]	Select the display mode. 00 = 4-bit dual scan display mode (STN) 01 = 4-bit single scan display mode (STN) 10 = 8-bit single scan display mode (STN) 11 = TFT LCD panel	00
BPPMODE	[4:1]	Select the BPP (Bits Per Pixel) mode. 0000 = 1 bpp for STN, Monochrome mode 0001 = 2 bpp for STN, 4-level gray mode 0010 = 4 bpp for STN, 16-level gray mode 0011 = 8 bpp for STN, color mode (256 color) 0100 = packed 12 bpp for STN, color mode (4096 color) 0101 = unpacked 12 bpp for STN, color mode (4096 color) 0110 = 16 bpp for STN, color mode (4096 color) 1000 = 1 bpp for TFT 1001 = 2 bpp for TFT 1010 = 4 bpp for TFT 1011 = 8 bpp for TFT 1100 = 16 bpp for TFT 1101 = 24 bpp for TFT	0000
ENVID	[0]	LCD video output and the logic enable/disable. 0 = Disable the video output and the LCD control signal. 1 = Enable the video output and the LCD control signal.	0

[27:18]为只读数据位，不需要设置； [17:8]用于设置 CLKVAL(像素时钟频率)，我们使用的是 TFT 屏，因此采用的公式是 VCLK = HCLK / [(CLKVAL+1) x 2]，其中 HCLK 为 100M。LCD 手册里面 Clock cycle 的要求范围为 5-12MHz 即可，即假设 VCLK=9，根据公式 9=100/[(CLKVAL+1)x2],算出 CLKVAL≈4.5=5。VCLK 为 plcdparams->time_seq.vclk，则 clkval =

HCLK/plcdparams->time_seq.vclk/2-1+0.5; [7]不用管，默认即可； [6:5]TFT lcd 配置为 0b11； [4:1]设置 bpp 模式，根据传入的`plcdparams->bpp`配置为相应的数值； [0]LCD 输出使能，先暂时关闭不输出；

寄存器

2：

LCD Control 2 Register

Register	Address	R/W	Description	Reset Value
LCDCON2	0X4D000004	R/W	LCD control 2 register	0x00000000

LCDCON2	Bit	Description	Initial State
VBPD	[31:24]	**TFT**: Vertical back porch is the number of inactive lines at the start of a frame, after vertical synchronization period. **STN**: These bits should be set to zero on STN LCD.	0x00
LINEVAL	[23:14]	**TFT/STN**: These bits determine the vertical size of LCD panel.	0000000000
VFPD	[13:6]	**TFT**: Vertical front porch is the number of inactive lines at the end of a frame, before vertical synchronization period. **STN**: These bits should be set to zero on STN LCD.	00000000
VSPW	[5:0]	**TFT**: Vertical sync pulse width determines the VSYNC pulse's high level width by counting the number of inactive lines. **STN**: These bits should be set to zero on STN LCD.	000000

对比 2440LCD 部分时序图和 LCD 时序图，得出两者之间关系，以后就可通过`plcdparams`传参数进来设置相关寄存器。 [31:24] : VBPD = tvb - 1 [23:14] : LINEVAL = line - 1 [13:6] : VFPD = tvf - 1 [5:0] : VSPW = tvp - 1

寄存器

3：

LCD Control 3 Register

Register	Address	R/W	Description	Reset Value
LCDCON3	0X4D000008	R/W	LCD control 3 register	0x00000000

LCDCON3	Bit	Description	Initial state
HBPD (TFT)	[25:19]	**TFT**: Horizontal back porch is the number of VCLK periods between the falling edge of HSYNC and the start of active data.	0000000
WDLY (STN)		**STN**: WDLY[1:0] bits determine the delay between VLINE and VCLK by counting the number of the HCLK. WDLY[7:2] are reserved. 00 = 16 HCLK, 01 = 32 HCLK, 10 = 48 HCLK, 11 = 64 HCLK	
HOZVAL	[18:8]	**TFT/STN**: These bits determine the horizontal size of LCD panel. HOZVAL has to be determined to meet the condition that total bytes of 1 line are 4n bytes. If the x size of LCD is 120 dot in mono mode, x=120 cannot be supported because 1 line consists of 15 bytes. Instead, x=128 in mono mode can be supported because 1 line is composed of 16 bytes (2n). LCD panel driver will discard the additional 8 dot.	00000000000
HFPD (TFT)	[7:0]	**TFT**: Horizontal front porch is the number of VCLK periods between the end of active data and the rising edge of HSYNC.	0X00
LINEBLANK (STN)		**STN**: These bits indicate the blank time in one horizontal line duration time. These bits adjust the rate of the VLINE finely. The unit of LINEBLANK is HCLK x 8. Ex) If the value of LINEBLANK is 10, the blank time is inserted to VCLK during 80 HCLK.	

[25:19] : HBPD = thb - 1 [18:8] : HOZVAL = 列 - 1 [7:0] : HFPD = thf - 1

寄存器

4：

LCD Control 4 Register

Register	Address	R/W	Description	Reset Value
LCDCON4	0X4D00000C	R/W	LCD control 4 register	0x00000000

LCDCON4	Bit	Description	Initial state
MVAL	[15:8]	**STN**: These bit define the rate at which the VM signal will toggle if the MMODE bit is set to logic '1'.	0X00
HSPW(TFT)	[7:0]	**TFT**: Horizontal sync pulse width determines the HSYNC pulse's high level width by counting the number of the VCLK.	0X00
WLH(STN)		**STN**: WLH[1:0] bits determine the VLINE pulse's high level width by counting the number of the HCLK. WLH[7:2] are reserved. 00 = 16 HCLK, 01 = 32 HCLK, 10 = 48 HCLK, 11 = 64 HCLK	

[7:0] : HSPW = thp - 1

寄存器

5：

LCD Control 5 Register

Register	Address	R/W	Description	Reset Value
LCDCON5	0X4D000010	R/W	LCD control 5 register	0x00000000

LCDCON5	Bit	Description	Initial state
Reserved	[31:17]	This bit is reserved and the value should be '0'.	0
VSTATUS	[16:15]	**TFT**: Vertical Status (read only). 00 = VSYNC 01 = BACK Porch 10 = ACTIVE 11 = FRONT Porch	00
HSTATUS	[14:13]	**TFT**: Horizontal Status (read only). 00 = HSYNC 01 = BACK Porch 10 = ACTIVE 11 = FRONT Porch	00
BPP24BL	[12]	**TFT**: This bit determines the order of 24 bpp video memory. 0 = LSB valid 1 = MSB Valid	0
FRM565	[11]	**TFT**: This bit selects the format of 16 bpp output video data. 0 = 5:5:5:1 Format 1 = 5:6:5 Format	0
INVVCLK	[10]	**STN/TFT**: This bit controls the polarity of the VCLK active edge. 0 = The video data is fetched at VCLK falling edge 1 = The video data is fetched at VCLK rising edge	0
INVVLINE	[9]	**STN/TFT**: This bit indicates the VLINE/HSYNC pulse polarity. 0 = Normal 1 = Inverted	0
INVVFRAME	[8]	**STN/TFT**: This bit indicates the VFRAME/VSYNC pulse polarity. 0 = Normal 1 = Inverted	0
INVVD	[7]	**STN/TFT**: This bit indicates the VD (video data) pulse polarity. 0 = Normal 1 = VD is inverted.	0
INVVDEN	[6]	**TFT**: This bit indicates the VDEN signal polarity. 0 = normal 1 = inverted	0
INVPWREN	[5]	**STN/TFT**: This bit indicates the PWREN signal polarity. 0 = normal 1 = inverted	0
INVLEND	[4]	**TFT**: This bit indicates the LEND signal polarity. 0 = normal 1 = inverted	0
PWREN	[3]	**STN/TFT**: LCD_PWREN output signal enable/disable. 0 = Disable PWREN signal 1 = Enable PWREN signal	0
ENLEND	[2]	**TFT**: LEND output signal enable/disable. 0 = Disable LEND signal 1 = Enable LEND signal	0
BSWP	[1]	**STN/TFT**: Byte swap control bit. 0 = Swap Disable 1 = Swap Enable	0
HWSWP	[0]	**STN/TFT**: Half-Word swap control bit. 0 = Swap Disable 1 = Swap Enable	0

用来设置引脚极性，设置 16bpp，设置内存中象素存放的格式 [12] : BPP24BL [11] : FRM565, 1-565 [10] : INVVCLK, 0 = The video data is fetched at VCLK falling edge [9] : HSYNC 是否反转 [8] : VSYNC 是否反转 [7] : INVVD, rgb 是否反转 [6] : INVVDEN [5] : INVPWREN [4] : INVLEND [3] : PWREN, LCD_PWREN output signal enable/disable [2] : ENLEND [1] : BSWP [0] : HWSWP

然后再设置 framebuffer 地址，先是

LCDSADDR1：

Register	Address	R/W	Description	Reset Value
LCDSADDR1	0X4D000014	R/W	**STN/TFT**: Frame buffer start address 1 register	0x00000000

LCDSADDR1	Bit	Description	Initial State
LCDBANK	[29:21]	These bits indicate A[30:22] of the bank location for the video buffer in the system memory. LCDBANK value cannot be changed even when moving the view port. LCD frame buffer should be within aligned 4MB region, which ensures that LCDBANK value will not be changed when moving the view port. So, care should be taken to use the malloc() function.	0x00
LCDBASEU	[20:0]	For dual-scan LCD : These bits indicate A[21:1] of the start address of the upper address counter, which is for the upper frame memory of dual scan LCD or the frame memory of single scan LCD. For single-scan LCD : These bits indicate A[21:1] of the start address of the LCD frame buffer.	0x000000

[29:21] : LCDBANK, A[30:22] of fb [20:0] : LCDBASEU, A[21:1] of fb 即用[29:0]表示起始地址的[30:1]。

LCDSADDR2：

Register	Address	R/W	Description	Reset Value
LCDSADDR2	0X4D000018	R/W	**STN/TFT**: Frame buffer start address 2 register	0x00000000

LCDSADDR2	Bit	Description	Initial State
LCDBASEL	[20:0]	For dual-scan LCD: These bits indicate A[21:1] of the start address of the lower address counter, which is used for the lower frame memory of dual scan LCD. For single scan LCD: These bits indicate A[21:1] of the end address of the LCD frame buffer. LCDBASEL = ((the frame end address) >>1) + 1 = LCDBASEU + (PAGEWIDTH+OFFSIZE) x (LINEVAL+1)	0x0000

[20:0] : LCDBASEL, A[21:1] of end addr 即 framebuffer 的结束地址。

最后还要设置相关引脚，包括背光控制引脚、LCD 专用引脚、电源控制引脚：

```
void jz2440_lcd_pin_init(void)
{
    /* 初始化引脚 : 背光引脚 */
    GPBCON &= ~0x3;
    GPBCON |= 0x01;
```

```
    /* LCD 专用引脚 */
    GPCCON = 0xaaaaaaaa;
    GPDCON = 0xaaaaaaaa;

    /* PWREN */
    GPGCON |= (3<<8);
}
```

LCD 所有寄存器的具体设置如下：

```
#define HCLK 100

void jz2440_lcd_pin_init(void)
{
    /* 初始化引脚 : 背光引脚 */
    GPBCON &= ~0x3;
    GPBCON |= 0x01;

    /* LCD 专用引脚 */
    GPCCON = 0xaaaaaaaa;
    GPDCON = 0xaaaaaaaa;

    /* PWREN */
    GPGCON |= (3<<8);
}


/* 根据传入的 LCD 参数设置 LCD 控制器 */
void s3c2440_lcd_controller_init(p_lcd_params plcdparams)
{
    int pixelplace;
    unsigned int addr;

    jz2440_lcd_pin_init();

    /* [17:8]: clkval, vclk = HCLK / [(CLKVAL+1) x 2]
     *                   9     = 100M /[(CLKVAL+1) x 2], clkval = 4.5 = 5
     *                 CLKVAL = 100/vclk/2-1
     * [6:5]: 0b11, tft lcd
     * [4:1]: bpp mode
     * [0]   : LCD video output and the logic enable/disable
     */
    int clkval = (double)HCLK/plcdparams->time_seq.vclk/2-1+0.5;
    int bppmode = plcdparams->bpp == 8  ? 0xb :\
                  plcdparams->bpp == 16 ? 0xc :\
                  0xd;  /* 0xd: 24bpp */
```

```
LCDCON1 = (clkval<<8) | (3<<5) | (bppmode<<1) ;

/* [31:24] : VBPD      = tvb - 1
 * [23:14] : LINEVAL = line - 1
 * [13:6]  : VFPD      = tvf - 1
 * [5:0]   : VSPW      = tvp - 1
 */
LCDCON2 =    ((plcdparams->time_seq.tvb - 1)<<24) | \
              ((plcdparams->yres - 1)<<14)            | \
             ((plcdparams->time_seq.tvf - 1)<<6)   | \
             ((plcdparams->time_seq.tvp - 1)<<0);

/* [25:19] : HBPD      = thb - 1
 * [18:8]  : HOZVAL   =  列  - 1
 * [7:0]   : HFPD      = thf - 1
 */
LCDCON3 =    ((plcdparams->time_seq.thb - 1)<<19) | \
             ((plcdparams->xres - 1)<<8)             | \
             ((plcdparams->time_seq.thf - 1)<<0);

/*
 * [7:0]   : HSPW      = thp - 1
 */
LCDCON4 =    ((plcdparams->time_seq.thp - 1)<<0);

 /*  用来设置引脚极性，设置 16bpp，设置内存中象素存放的格式
  * [12] : BPP24BL
 * [11] : FRM565, 1-565
 * [10] : INVVCLK, 0 = The video data is fetched at VCLK falling edge
 * [9]   : HSYNC 是否反转
 * [8]   : VSYNC 是否反转
 * [7]   : INVVD, rgb 是否反转
 * [6]   : INVVDEN
 * [5]   : INVPWREN
 * [4]   : INVLEND
 * [3]   : PWREN, LCD_PWREN output signal enable/disable
 * [2]   : ENLEND
 * [1]   : BSWP
 * [0]   : HWSWP
 */

pixelplace = plcdparams->bpp == 24 ? (0) : |\
               plcdparams->bpp == 16 ? (1) : |\
               (1<<1);   /* 8bpp */
```

```
    LCDCON5 = (plcdparams->pins_pol.vclk<<10) |\
              (plcdparams->pins_pol.rgb<<7)    |\
              (plcdparams->pins_pol.hsync<<9) |\
              (plcdparams->pins_pol.vsync<<8) |\
             (plcdparams->pins_pol.de<<6)     |\
             (plcdparams->pins_pol.pwren<<5) |\
             (1<<11) | pixelplace;

    /* framebuffer 地址 */
    /*
     * [29:21] : LCDBANK, A[30:22] of fb
     * [20:0]  : LCDBASEU, A[21:1] of fb
     */
    addr = plcdparams->fb_base & ~(1<<31);
    LCDSADDR1 = (addr >> 1);

    /*
     * [20:0] : LCDBASEL, A[21:1] of end addr
     */
    addr = plcdparams->fb_base +
plcdparams->xres*plcdparams->yres*plcdparams->bpp/8;
    addr >>=1;
    addr &= 0x1fffff;
    LCDSADDR2 = addr;//
}

void s3c2440_lcd_controller_enalbe(void)
{
    /* 背光引脚 : GPB0 */
    GPBDAT |= (1<<0);

    /* pwren    : 给 LCD 提供 AVDD  */
    LCDCON5 |= (1<<3);

    /* LCDCON1'BIT 0 : 设置 LCD 控制器是否输出信号 */
    LCDCON1 |= (1<<0);
}

void s3c2440_lcd_controller_disable(void)
{
    /* 背光引脚 : GPB0 */
    GPBDAT &= ~(1<<0);

    /* pwren    : 给 LCD 提供 AVDD  */
```

```
    LCDCON5 &= ~(1<<3);

    /* LCDCON1'BIT 0 : 设置 LCD 控制器是否输出信号 */
    LCDCON1 &= ~(1<<0);
}
```

这就完成了`s3c2440_lcd_controller.c`的编写，后面只需要向`s3c2440_lcd_controller_init()`传入构造好的参数即可。

第 006 节_编程_LCD 设置

前面编写了`s3c2440_lcd_controller.c`，以后我们只需往里面传入参数即可控制 LCD 控制器，对于我们的 4.3 寸 LCD，配合 LCD 手册时序的介绍，相关的设置如下:

Item	Symbol	Values			Unit	Remark
		Min.	Typ.	Max.		
Clock cycle	1/tc	5	9.00	12	MHz	
Hsync cycle	$1/f_H$	-	17.14	-	KHz	
Vsync cycle	1/fv	59.94	-	-	Hz	
Horizontal signal	t_h	-	525	-	CLK	Note 1
Horizontal display period	t_{hd}	-	480	-	CLK	
Horizontal Front porch	t_{hf}	2	-	-	CLK	Note 2
Horizontal Pulse width	t_{hp}	2	41	-	CLK	Note 2
Horizontal Back porch	t_{hb}	2	-	-	CLK	Note 2
Vertical cycle	t_v	-	286	-	H	
Vertical display period	t_{vd}	-	272	-	H	
Vertical Front porch	t_{vf}	2	2	-	H	
Vertical Pulse width	t_{vp}	2	10	-	H	
Vertical Back porch	t_{vb}	2	2	-	H	
DISP Setup Time	t_{diss}	10	-	-	ns	
DISP Hold Time	t_{dish}	10	-	-	ns	
Clock Period	PW CLK	66.7	-	-	ns	
Clock Pulse High Period	PWH	26.7	-	-	ns	
Clock Pulse Low Period	PWL	26.7	-	-	ns	
Hsync Setup Time	t_{hs}	10	-	-	ns	
Hsync Hold Time	t_{hh}	10	-	-	ns	
Data Setup Time	t_{ds}	10	-	-	ns	
Data Hold Time	t_{dh}	10	-	-	ns	
DE Setup Time	t_{des}	10	-	-	ns	
DE Hold Time	t_{deh}	10	-	-	ns	
Vsync Setup Time	t_{vns}	10	-	-	ns	
Vsync Hold Time	t_{vnh}	10	-	-	ns	

Note 1: t_{hd}=480CLK, t_{hf}= 2CLK, t_{hp}= 41CLK, t_{hb}= 2CLK
525CLK=480CLK + 2CLK + 41CLK + 2CLK
Note 2: t_{hf}+ t_{hp}+ t_{hb}> 44 CLK

```
#define LCD_FB_BASE 0x33c00000

lcd_params lcd_4_3_params = {
    .name = "lcd_4.3"
    .pins_polarity = {
        .de    = NORMAL, /* normal: 高电平时可以传输数据 */
        .pwren = NORMAL,    /* normal: 高电平有效 */
        .vclk  = NORMAL, /* normal: 在下降沿获取数据 */
        .rgb   = NORMAL, /* normal: 高电平表示 1 */
        .hsync = INVERT,    /* normal: 高脉冲 */
        .vsync = INVERT,   /* normal: 高脉冲 */
    },
    .time_sequence = {
        /* 垂直方向 */
        .tvp=    10, /* vysnc 脉冲宽度 */
        .tvb=    2,  /* 上边黑框, Vertical Back porch */
        .tvf=    2,  /* 下边黑框, Vertical Front porch */

        /* 水平方向 */
        .thp=    41, /* hsync 脉冲宽度 */
        .thb=    2,  /* 左边黑框, Horizontal Back porch */
        .thf=    2,  /* 右边黑框, Horizontal Front porch */

        .vclk=   9,  /* MHz */
    },
    .xres = 480,
    .yres = 272,
    .bpp  = 16,
    .fb_base = LCD_FB_BASE,
};
```

完成了 lcd 控制器和参数的代码，现在还需要一个管理的中间层将两者连在一起。 我们用`lcd_controller.c`管理`s3c2440_lcd_controller.c`，它向上接受传入的 LCD 参数，向下传给对应的 LCD 控制器。`lcd_controller.c`管理下级的控制器的思路如下： a. 用数组保存下面各种 lcd_controller； b. 提供`register_lcd_controller`给下面的代码设置数组； c. 提供`select_lcd_controller(name)`给上面的代码选择某个`lcd_controller`;

`lcd_controller.c`代码如下：

```
#define LCD_CONTROLLER_NUM 10

static p_lcd_controller p_array_lcd_controller[LCD_CONTROLLER_NUM];
static p_lcd_controller g_p_lcd_controller_selected;

int register_lcd_controller(p_lcd_controller plcdcon)
```

```
{
    int i;
    for (i = 0; i < LCD_CONTROLLER_NUM; i++)
    {
        if (!p_array_lcd_controller[i])
        {
            p_array_lcd_controller[i] = plcdcon;
            return i;
        }
    }
    return -1;
}

int select_lcd_controller(char *name)
{
    int i;
    for (i = 0; i < LCD_CONTROLLER_NUM; i++)
    {
        if (p_array_lcd_controller[i]
&& !strcmp(p_array_lcd_controller[i]->name, name))
        {
            g_p_lcd_controller_selected = p_array_lcd_controller[i];
            return i;
        }
    }
    return -1;
}


/* 向上: 接收不同 LCD 的参数
 * 向下: 使用这些参数设置对应的 LCD 控制器
 */

int lcd_controller_init(p_lcd_params plcdparams)
{
    /* 调用所选择的 LCD 控制器的初始化函数 */
    if (g_p_lcd_controller_selected)
    {
        g_p_lcd_controller_selected->init(plcdparams);
        return 0;
    }
    return -1;
}
```

```
void lcd_contoller_add(void)
{
    s3c2440_lcd_contoller_add();
}
```

同时，在`s3c2440_lcd_controller.c`里注册控制器：

```
void s3c2440_lcd_contoller_add(void)
{
    register_lcd_controller(&s3c2440_lcd_controller);
}
```

这样，`s3c2440_lcd_controller.c`里的`register_lcd_controller()`将自己放在`p_array_lcd_controller[]`这个数组，然后上层的`lcd_controller.c`调用`select_lcd_controller()`传入要选择的 LCD 控制器，然后在数组里面找到名字名字匹配的 LCD 控制器进行相应的初始化。

同理，也通过`lcd.c`去管理`lcd_4.3.c`,思路如下： a. 有一个数组存放各类 lcd 的参数； b. 有一个 register_led 给下面的 lcd 程序来设置数组； c. 有一个 select_lcd，供上层选择某款 LCD；

参考前面的`lcd_controller.c`编辑`lcd_controller.c`如下：

```
#define LCD_NUM 10

static p_lcd_params p_array_lcd[LCD_NUM];
static p_lcd_params g_p_lcd_selected;

int register_lcd(p_lcd_params plcd)
{
    int i;
    for (i = 0; i < LCD_NUM; i++)
    {
        if (!p_array_lcd[i])
        {
            p_array_lcd[i] = plcd;
            return i;
        }
    }
    return -1;
}

int select_lcd(char *name)
{
    int i;
    for (i = 0; i < LCD_NUM; i++)
    {
```

```
        if (p_array_lcd[i] && !strcmp(p_array_lcd[i]->name, name))
        {
            g_p_lcd_selected = p_array_lcd[i];
            return i;
        }
    }
    return -1;
}
```

在`lcd_4.3.c`里面把 lcd 参数注册进去： ``` void lcd_4_3_add(void) { register_lcd(&lcd_4_3_params); } ```

以后只需要在`lcd.c`里面选择某款 lcd 和某款 lcd 控制器即可，底层的只管添加种类即可。 在`lcd.c`里面添加初始化函数如下：

```
int lcd_init(void)
{
    /* 注册 LCD */
    lcd_4_3_add();

    /* 注册 LCD 控制器 */
    lcd_contoller_add();

    /* 选择某款 LCD */
    select_lcd("lcd_4.3");

    /* 选择某款 LCD 控制器 */
    select_lcd_controller("s3c2440");

    /* 使用 LCD 的参数, 初始化 LCD 控制器 */
    lcd_controller_init(g_p_lcd_selected);
}
```

第 007 节_编程_简单测试

首先向 lcd_test.c 里面添加一个测试函数 lcd_test()，用于向 framebuffer 写数据，所需步骤如下： > a. 初始化 LCD > b. 使能 LCD > c. 获取 LCD 参数: fb_base, xres, yres, bpp > d. 往 framebuffer 中写数据

a. 初始化 LCD

```
lcd_init();
```

b. 使能 LCD

```
lcd_enable();
```

该函数实际调用的是`lcd_controller_enable()`

c. 获取 LCD 参数: fb_base, xres, yres, bpp 只有获取到 LCD 的参数信息，才能根据这些信息进行相应显示。

```
get_lcd_params(&fb_base, &xres, &yres, &bpp);
```

该函数是在`lcd.c`里面实现：

```
void get_lcd_params(unsigned int *fb_base, int *xres, int *yres, int *bpp)
```

```
{
    *fb_base = g_p_lcd_selected->fb_base;
    *xres = g_p_lcd_selected->xres;
    *yres = g_p_lcd_selected->yres;
    *bpp = g_p_lcd_selected->bpp;
}
```

d. 往 framebuffer 中写数据 假设现在 BPP=16，想让全屏显示红色，就需要从 framebuffer 基地址开始一直填充对应的颜色数据。 对于 16BPP，RGB=565，想显示红色，即[15:11]全为 1 表示红色，[10:5]全为 0 表示无绿色，[4:0]全为 0 表示无蓝色，0b1111100000000000=0xF800。 以基地址为起点，分别以 xres 和 yres 为边界，依次填充颜色。

```
p = (unsigned short *)fb_base;
for (x = 0; x < xres; x++)
    for (y = 0; y < yres; y++)
        *p++ = 0xf800;
```

编写好程序后，修改 Makefile:

```
objs = start.o led.o uart.o init.o nand_flash.o main.o exception.o interrupt.o timer.o nor_flash.o my_printf.o string_utils.o lib1funcs.o

objs += lcd/font.o
objs += lcd/framebuffer.o
objs += lcd/geometry.o
objs += lcd/lcd.o
objs += lcd/lcd_4.3.o
objs += lcd/lcd_controller.o
objs += lcd/lcd_test.o
objs += lcd/s3c2440_lcd_controller.o

all: $(objs)
    #arm-linux-ld -Ttext 0 -Tdata 0x30000000    start.o led.o uart.o init.o main.o -o sdram.elf
    arm-linux-ld -T sdram.lds $^ -o sdram.elf
    arm-linux-objcopy -O binary -S sdram.elf sdram.bin
    arm-linux-objdump -D sdram.elf > sdram.dis
clean:
    rm *.bin *.o *.elf *.dis

%.o : %.c
    arm-linux-gcc -march=armv4 -c -o $@ $<

%.o : %.S
    arm-linux-gcc -march=armv4 -c -o $@ $<
```

然后把工程文件放到虚拟机上交叉编译，根据编译提示结果进行对应修改。 常见问题包括：头文件未添加、数据类型错误等。

同理，假如现在是 24BPP，即 RGB:888，每个颜色占 8 位，一共占据 24 位。虽然颜色数据只占据 24 位，但实际中是占据的 32 位(1 字节)方便存储计算，即[23:0]存放的是数据，[31：24]空闲无数据。 对于 32BPP，大多数情况下和 24BPP 差不多的，即 RGB:888，每个颜色占 8 位，一共占据 24 位。因此想依次显示红绿蓝，代码如下：

```
/* 0xRRGGBB */
  /* red */
p2 = (unsigned int *)fb_base;
for (x = 0; x < xres; x++)
    for (y = 0; y < yres; y++)
        *p2++ = 0xff0000;

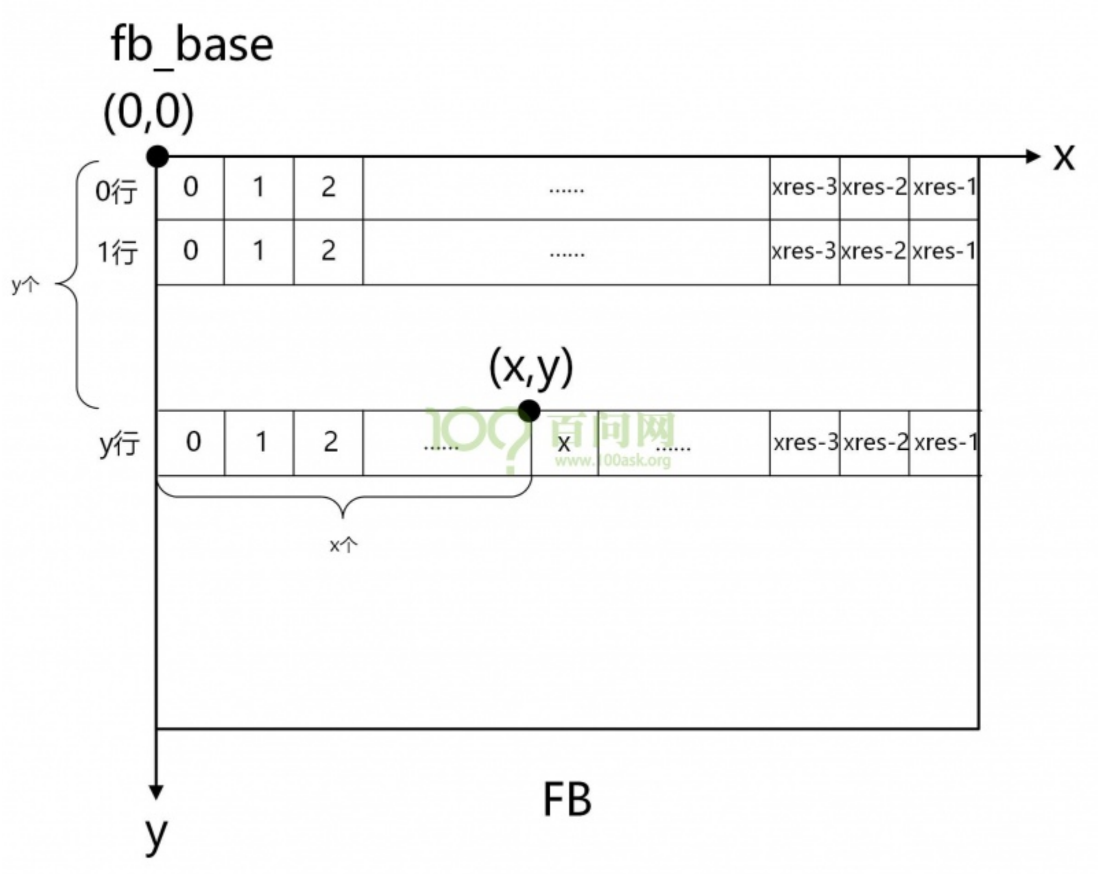
/* green */
p2 = (unsigned int *)fb_base;
for (x = 0; x < xres; x++)
    for (y = 0; y < yres; y++)
        *p2++ = 0x00ff00;

/* blue */
p2 = (unsigned int *)fb_base;
for (x = 0; x < xres; x++)
    for (y = 0; y < yres; y++)
        *p2++ = 0x0000ff;
```

之前我们讲数据是 8BPP 的时候，可以通过调色板转成 16BPP，在 LCD 上显示出相应颜色。 那前面的 24BPP、32BPP 是怎样在 只能接收 16BPP(硬件上只有 16 根数据线)的 LCD 上显示的呢？ 这是因为在使用 24BPP 时，发出的 8 条红色，8 条绿色，8 条蓝色数据，只用了高 5 条红色，高 6 条绿色，高 5 条蓝色与 LCD 相连。

第 008 节_编程_画点线圆

本节将在 LCD 上画点画圆，无论是何种图形，都是基于点来构成的，因此我们需要先实现画点。 在前面**第 003 节_编程_框架与准备**所讲的框架里，计划的是在`farmebuffer.c`实现画点，在`geomentry.c`实现画线、画圆，`font.c`实现写字。 我们先在`farmebuffer.c`实现画点，一个点(x，y)在 FB 中的位置如图：



可以得出其计算公式：

(x，y)像素起始地址=fb_base+(xres*bpp/8)*y + y*bpp/8

同时利用 yres 来分辨 y 方向的边界。 因此，需要先从 LCD 中获取参数：fb_base、xres、yres、bpp;

```
static unsigned int fb_base;
static int xres, yres, bpp;

void fb_get_lcd_params(void)
{
    get_lcd_params(&fb_base, &xres, &yres, &bpp);
}
```

再实现画点操作：

```
void fb_put_pixel(int x, int y, unsigned int color)
{
    unsigned char   *pc;    /* 8bpp */
    unsigned short *pw;    /* 16bpp */
    unsigned int     *pdw; /* 32bpp */

    unsigned int pixel_base = fb_base + (xres * bpp / 8) * y + x * bpp / 8;

    switch (bpp)
    {
```

```
        case 8:
            pc = (unsigned char *) pixel_base;
            *pc = color;
            break;
        case 16:
            pw = (unsigned short *) pixel_base;
            *pw = convert32bppto16bpp(color);
            break;
        case 32:
            pdw = (unsigned int *) pixel_base;
            *pdw = color;
            break;
    }
}
```

对于 8PP，每个像素只占据 8 位(1 字节)，因此采用`unsigned char`类型；对于 16PP，每个像素只占据 16 位(2 字节)，因此采用`unsigned short`类型； 对于 32PP，每个像素只占据 32 位(4 字节)，因此采用`unsigned int`类型；

再根据传入的 x,y 坐标，计算出对应的显存位置。

根据 BPP 的不同，修改相应位的显存数据。 传入的颜色数据一般都是 32bit 的，即格式为：0x00RRGGBB, 对于 8PP，通过的是调色板索引实现的，这个后续再讲解，直接`*pc = color`即可。 对于 16PP，需要进行颜色转换。 对于 32PP，大小刚好对应，直接`*pc = color`即可。

使用`convert32bppto16bpp()`函数进行颜色数据转换：

```
unsigned short convert32bppto16bpp(unsigned int rgb)
{
    int r = (rgb >> 16)& 0xff;
    int g = (rgb >> 8) & 0xff;
    int b = rgb & 0xff;

    /* rgb565 */
    r = r >> 3;
    g = g >> 2;
    b = b >> 3;

    return ((r<<11) | (g<<5) | (b));
}
```

先分别取出 RGB，再相应的清除低位数据，实现将 RGB888 变为 RGB565，最后再组成`unsigned short`类型数据返回。

画圆画线的具体原理不是我们的主要内容，我们直接百度“C 语言 LCD 画圆”可以得到相关的实现代码，比如这篇博客：http://blog.csdn.net/p1126500468/article/details/50428613 新建一个`geometry.c`，复制博客中代码，替换里面的描点显示函数即可。

最后在主函数测试程序里，加上画圆画线的测试代码：

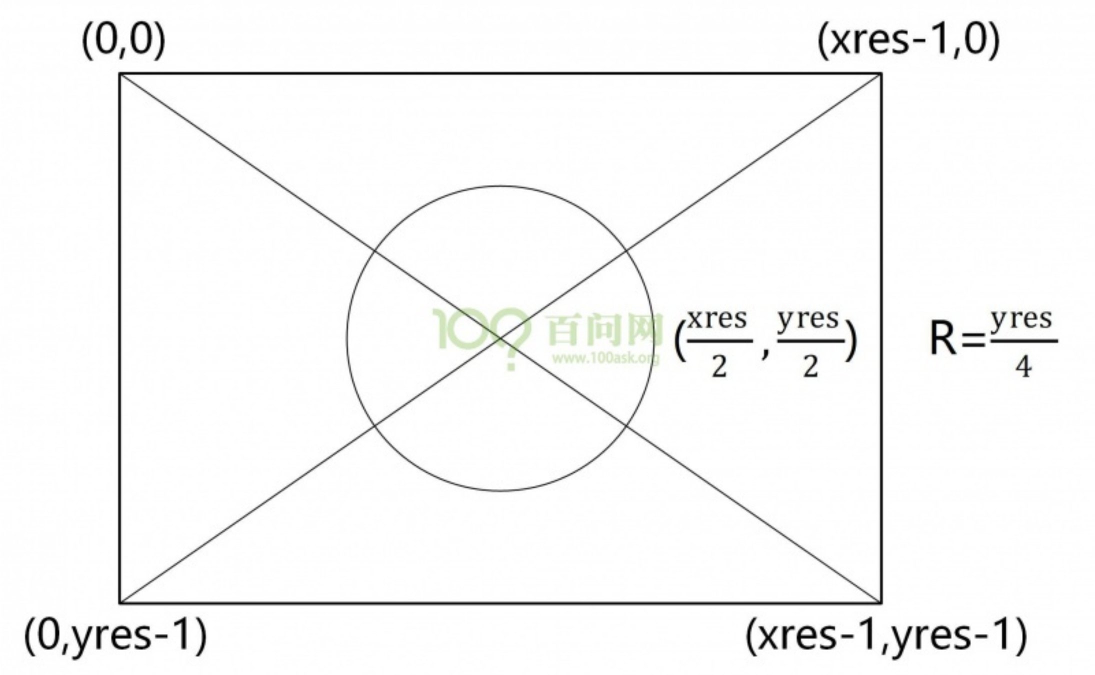
```
    /*  画线  */
```

```
draw_line(0, 0, xres - 1, 0, 0xff0000);
draw_line(xres - 1, 0, xres - 1, yres - 1, 0xffff00);
draw_line(0, yres - 1, xres - 1, yres - 1, 0xff00aa);
draw_line(0, 0, 0, yres - 1, 0xff00ef);
draw_line(0, 0, xres - 1, yres - 1, 0xff4500);
draw_line(xres - 1, 0, 0, yres - 1, 0xff0780);

delay(1000000);

/* 画圆 */
draw_circle(xres/2, yres/2, yres/4, 0xff00);
```

希望在 LCD 上显示如下图形，由 6 条线和一个圆组成。 将线的起始坐标作为参数传入画线函数。 将圆心和半径作为参数传入画圆函数。



第 009 节_编程_显示文字

文字也是由点构成的，一个个点组成的点阵，宏观的来看，就是文字。 可以参考 Linux 内核源码中的相关操作，在内核中搜索“font”，打开`font_8x16.c`，可以看到里面的 A 字符内容如下：

```
/* 65 0x41 'A' */
0x00, /* 00000000 */
0x00, /* 00000000 */
0x10, /* 00010000 */
0x38, /* 00111000 */
0x6c, /* 01101100 */
0xc6, /* 11000110 */
0xc6, /* 11000110 */
```

```
0xfe, /* 11111110 */
0xc6, /* 11000110 */
0xc6, /* 11000110 */
0xc6, /* 11000110 */
0xc6, /* 11000110 */
0x00, /* 00000000 */
0x00, /* 00000000 */
0x00, /* 00000000 */
0x00, /* 00000000 */
```

根据这些数据，在一个 8*16 的区域里，将为 1 的点显示出来，为 0 的则不显示，最终将呈现一个字母“A”。

新建一个`font.c`，根据字母的点阵在 LCD 上描画文字，需要的步骤如下：a. 根据带显示的字符的 ascii 码在 fontdata_8x16 中得到点阵数据 b. 根据点阵来设置对应象素的颜色 c. 根据点阵的某位决定是否描颜色

```
void fb_print_char(int x, int y, char c, unsigned int color)
{
    int i, j;

    /* 根据 c 的 ascii 码在 fontdata_8x16 中得到点阵数据 */
    unsigned char *dots = &fontdata_8x16[c * 16];

    unsigned char data;
    int bit;

    /* 根据点阵来设置对应象素的颜色 */
    for (j = y; j < y+16; j++)
    {
        data = *dots++;
        bit = 7;
        for (i = x; i < x+8; i++)
        {
            /* 根据点阵的某位决定是否描颜色 */
            if (data & (1<<bit))
                fb_put_pixel(i, j, color);
            bit--;
        }
    }
}
```

在`font_8x16.c`里面，每个字符占据 16 位，因此想要根据 ascii 码找到对应的点阵数据，需要对应的乘 16，再取地址，得到该字符的首地址。 再根据每个点阵数据每位是否为 1，来调用描点函数`fb_put_pixel()`。这样，依次显示 16 个点阵数据，获得字符图形。

同样的，在显示之前，还需要获取 LCD 参数：

```
extern const unsigned char fontdata_8x16[];
```

```
/* 获得 LCD 参数 */
static unsigned int fb_base;
static int xres, yres, bpp;

void font_init(void)
{
    get_lcd_params(&fb_base, &xres, &yres, &bpp);
}
```

如果想显示字符串，那就在每显示完一个字符后，x 轴加 8 即可，同时考虑是否超出屏幕显示范围进行换行处理：

```
/* "abc\n\r123" */
void fb_print_string(int x, int y, char* str, unsigned int color)
{
    int i = 0, j;

    while (str[i])
    {
        if (str[i] == '\n')
            y = y+16;
        else if (str[i] == '\r')
            x = 0;

        else
        {
            fb_print_char(x, y, str[i], color);
            x = x+8;
            if (x >= xres) /* 换行 */
            {
                x = 0;
                y = y+16;
            }
        }
        i++;
    }
}
```

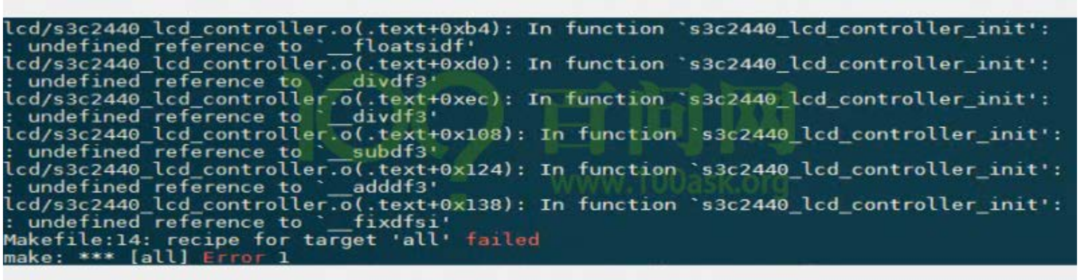
最后在在主函数里，加上显示字符串的函数，传入希望显示的字符串。

第 010 节_编程_添加除法

在`s3c2440_lcd_controller.c`里，以前我们使用如下除法：

```
int clkval = (double)HCLK/plcdparams->time_seq.vclk/2-1+0.5;
```

编译的时候会提示出
错：



我们的`lib1funcs.S`里面是有除法的，但除法的功能不够强，将前面除法计算代码中的`double`改成精度更小的`float`还是不行。

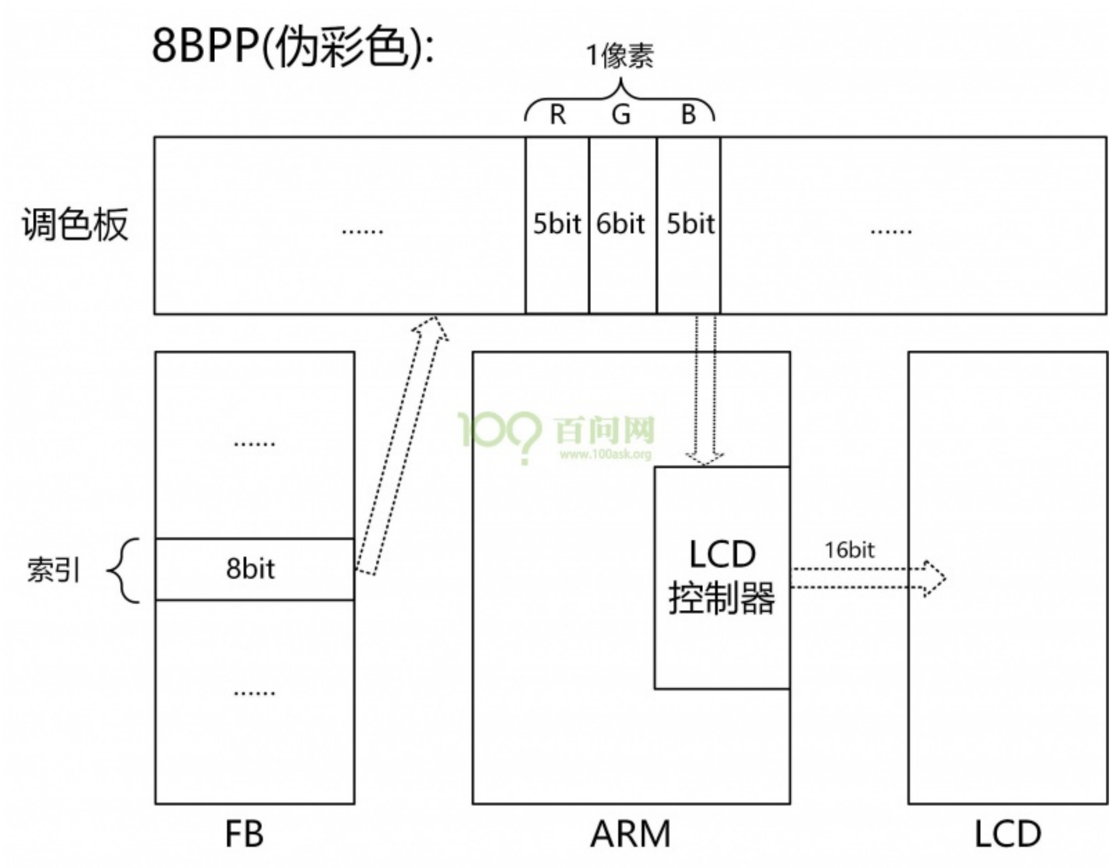
对于未实现的函数： a. 去 uboot 中查找 b. 去内核源码中查找 c. 去库函数中查找(一般来说编译器自带有很多库)

在库函数中查找步骤： a. 输入命令`arm-linux-gcc -v`，查看当前使用的交叉编译工具链； b. 输入命令`echo $PATH`，在环境变量中找到当前使用的交叉编译工具链所在的路径； c. 进入交叉编译工具链所在目录搜索相关函数，例如`grep "__floatsisf" * -nR`； d. 提取出其中的静态库(.a 后缀文件)，复制文件到代码文件； e. 修改 Makefile，依次尝试加入的每个静态库，直至编译成功；

注意:如果你更换了编译器，需要自己去编译器目录里找出对应的`libgcc.a`;有可能有多个`libgcc.a`，逐个尝试；

第 011 节_编程_使用调色板

前面我们写的程序都是采用的 16BPP 或者 24BPP(也就是 32BPP)，假如我们要使用 8PP，就得使用调色板。 如图所示 8PP 工作原理示意图，在 FB 只存放 8bit 得每个像素索引，根据这个索引，在去去调色板找到对应的数据传给 LCD 控制器，再通过电子枪显示出来。 调色板里面有 2^8(256)个颜色数据，每个颜色数据为 16bit，表示一种颜
色。



在硬件上，我们要初始化这个调色板，才能通过索引得到颜色。

根据第三节的软件框架，调色板的初始化应该放在`s3c2440_lcd_controller.c`里面。 在 lcd_controller 结构体里添加调色板初始化函数：

```
struct lcd_controller s3c2440_lcd_controller = {
    .name       = "s3c2440",
    .init       = s3c2440_lcd_controller_init,
    .enable     = s3c2440_lcd_controller_enalbe,
    .disable = s3c2440_lcd_controller_disable,
    .init_palette = s3c2440_lcd_controller_init_palette,
};
```

调色板对应一块内存，我们需要找到他的位置和格式。 800px 从芯片手册了解到，其起始地址为 0x4D000400，且设置调色板前，需要关闭 LCD 控制器。

```
void s3c2440_lcd_controller_init_palette(void)
{
    volatile unsigned int *palette_base =   (volatile unsigned int *)0x4D000400;
    int i;

    int bit = LCDCON1 & (1<<0);

    /* LCDCON1'BIT 0 : 设置 LCD 控制器是否输出信号  */
    if (bit)
```

```c
        LCDCON1 &= ~(1<<0);

    for (i = 0; i < 256; i++)
    {
        /* 低 16 位 : rgb565 */
        *palette_base++ = i;
    }

    if (bit)
        LCDCON1 |= (1<<0);
}
```

设置调色板前，先判断 LCD 控制器是否打开，如果打开了就先关闭，且设置完成后再打开。 再网上搜索了一下，没有找到调色板数据数组，这里作为实验，就随便设置，让其为 i。 我们让调色板数据等于 i，0-255，只占据 8 位，最后的颜色范围为 B5G3R0，因此会比较偏蓝。

修改`lcd_4.3.c`，将 BPP 改为 8， 再修改`lcd_controller.c`的初始化函数，加入调色板初始化函数。

```c
int lcd_controller_init(p_lcd_params plcdparams)
{
    /* 调用所选择的 LCD 控制器的初始化函数 */
    if (g_p_lcd_controller_selected)
    {
        g_p_lcd_controller_selected->init(plcdparams);
        g_p_lcd_controller_selected->init_palette();
        return 0;
    }
    return -1;
}
```

再修改`lcd_test.c`，加入 bpp=8 的情况：

```c
    if (bpp == 8)
    {
        /* 让 LCD 输出整屏的红色 */

        /* bpp: palette[12] */

        p0 = (unsigned char *)fb_base;
        for (x = 0; x < xres; x++)
            for (y = 0; y < yres; y++)
                *p0++ = 12;

        /* palette[47] */
        p0 = (unsigned char *)fb_base;
        for (x = 0; x < xres; x++)
            for (y = 0; y < yres; y++)
```

```
                *p0++ = 47;

        /* palette[88] */
        p0 = (unsigned char *)fb_base;
        for (x = 0; x < xres; x++)
            for (y = 0; y < yres; y++)
                *p0++ = 88;

        /* palette[0] */
        p0 = (unsigned char *)fb_base;
        for (x = 0; x < xres; x++)
            for (y = 0; y < yres; y++)
                *p0++ = 0;

    }
```

随便让其全屏显示某个颜色。

最后在画线画圆，显示文字的函数里，修改下颜色：

```
    /* 画线 */
    draw_line(0, 0, xres - 1, 0, 0x23ff77);
    draw_line(xres - 1, 0, xres - 1, yres - 1, 0xffff);
    draw_line(0, yres - 1, xres - 1, yres - 1, 0xff00aa);
    draw_line(0, 0, 0, yres - 1, 0xff00ef);
    draw_line(0, 0, xres - 1, yres - 1, 0xff45);
    draw_line(xres - 1, 0, 0, yres - 1, 0xff0780);

    delay(1000000);

    /* 画圆 */
    draw_circle(xres/2, yres/2, yres/4, 0xff);

    /* 输出文字 */
    fb_print_string(10, 10, "www.100ask.net\n\r100ask.taobao.com", 0xff);
```

上面的颜色数据，其实只有低两位有效，因为前面的调色板映射范围是 0-255。

第 018 课 ADC 和触摸屏

第 001 节_ADC 硬件原理

模数转换器即 A/D 转换器，或简称 ADC，通常是指一个将模拟信号转变为数字信号的电子元件。

通常的模数转换器是把经过与标准量比较处理后的模拟量转换成以二进制数值表示的离散信号的转换器。

故任何一个模数转换器都需要一个参考模拟量作为转换的标准，比较常见的参

考标准为最大的可转换信号大小。

而输出的数字量则表示输入信号相对于参考信号的大小。如图，是把可变电阻上的电压值变换的模拟信号通过 ADC 转换，输出数字信号。



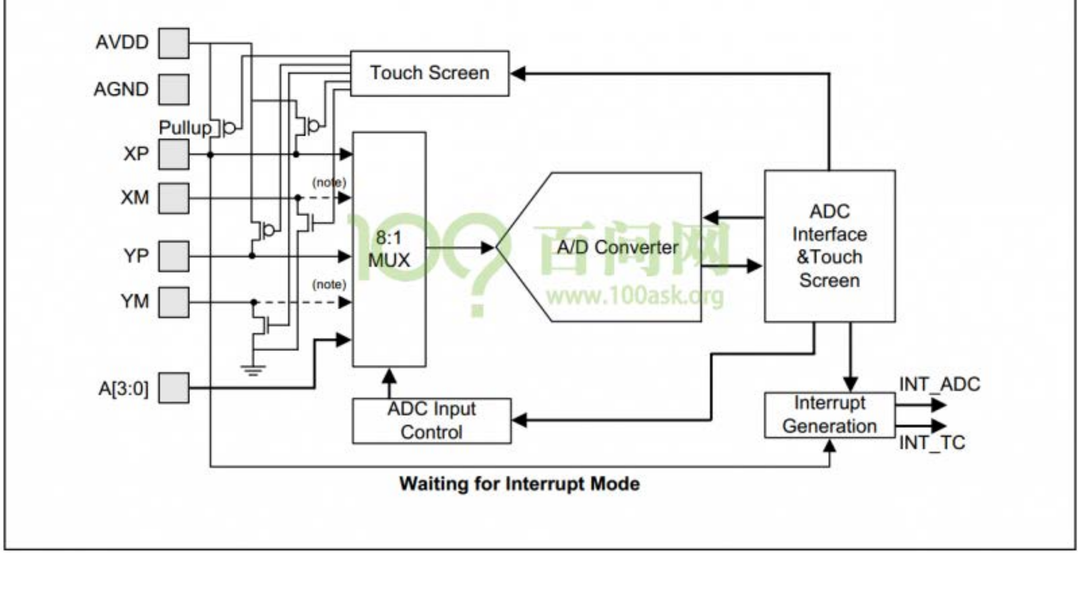
对于数字信号我们需要得到它的几个属性

- 用多少位来存储这个数据（假设 10bit）。
- 最大值 0b1111111111
- 它对应的电压是多少伏（模拟信号输入的最大值是多少）我们就可以根据模拟信号(电压)的最大值，来计算出对应的数值。
- 采样/转换速度。

对于程序员，我们不关心 ADC 的内部机制，我们只关心：

- 怎么启动 ADC
- 启动之后怎么得到数据，

总之：我们都是通过寄存器操作的。



从图 1-1-1 可以看出 ADC 有 8 个多路选择器，显然，以后我们写程序的时候，我们可以 8 个多路选择之一，　下面是编写程序要做的步骤：

1. 确定是哪一路信号：设置 8：1MUX，选择要测量哪一个引脚，（看原理图选择要测量的引脚）
2. 设置工作时钟（从工作室中，可以算出转换一次，需要多长时间）
3. 启动
4. 读状态，判断 ADC 转换是否成功。
5. 读数据

ADC 寄存器介绍

1，ADC 控制寄存器（ADCCON） ADCCON 控制寄存器，用于标志转换是否完成，控制是否使能预分频器，输入通道选择，工作模式，ADC 是否启动。它的各位含义如下图所示。

寄存器	地址	R/W	描述	ADC 控制寄存器
ADCCON	0x5800000	R/W	ADC 控制寄存器	0x3FC4

ADCCON	位	描述	初始状态
ECFLG	[15]	转换结束标志位（只读） 0 = A/D 正在转换 1 = A/D 转换已结束	0
PRSCEN	[14]	A/D 转换器预分频器使能 0 = 禁止　　1 = 使能	0
PRSCVL	[13:6]	A/D 转换器预分频值 数值范围： 0 至 255 注意：ADC 频率应该设置为低于 PCLK 的 1/5。(例如 PCLK=10MHz，则 ADC 频率<2MHz)	0xFF
SEL_MUX	[5:3]	模拟输入通道选择 000 = AIN 0 001 = AIN 1 010 = AIN 2 011 = AIN 3 100 = YM 101 = YP 110 = XM 111 = XP	0
STDBM	[2]	待机模式选择 0 = 正常工作模式　1 = 待机模式	1
READ_ START	[1]	读启动 A/D 转换 0 = 禁止读启动操作 1 = 使能读启动操作	0
ENABLE_START	[0]	使能 A/D 转换启动。如果 READ_START 为使能，则此值无效 0 = 无操作　1 = A/D 转换启动且此位在启动后被清零。	0

2，ADC 启动延时寄存器（ADCDLY） ADCDLY 启动延时寄存器用于启动或初始化延时寄存器。它的各位含义如下图所示

寄存器	地址	R/W	描述	ADC 控制寄存器
ADCDLY	0x5800008	R/W	ADC 启动或初始化延时寄存器	0x00ff

ADCDLY	位	描述	初始状态
DELAY	[15:0]	正常转换模式、XY 方向模式、自动方向模式 →ADC 转换启动延时值。 注意：不要使用 0 这个值（0x0000）	0x0ff

3，ADC 转换数据寄存器（ADCDAT0） ADCDAT0 转换数据寄存器，本节中只用到该寄存器的前 10 位(用于保存转换后的结果)。

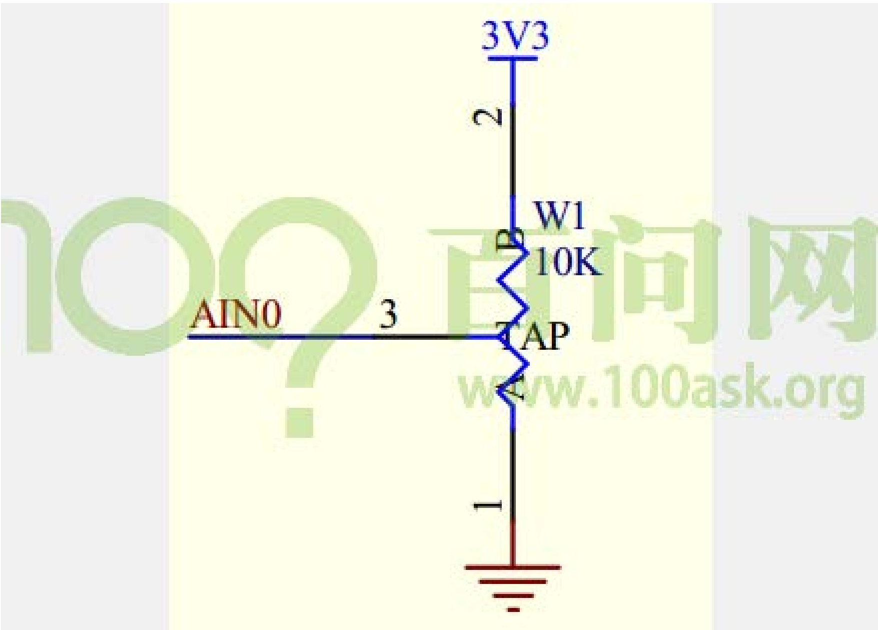
寄存器	地址	R/W	描述	ADC 控制寄存器
ADCDAT0	0x580000C	R	ADC 转换数据寄存器	-

ADCDAT0	位	描述	初始状态
UPDOWN	[15]	等待中断模式中笔尖的起落状态 0 = 笔尖落下态 1 = 笔尖抬起态	-
AUTO_PST	[14]	自动顺序 X 方向和 Y 方向转换 0 = 正常 ADC 转换 1 = 顺序 X 方向、Y 方向测量	-
XY_PST	[13:12]	手动 X 方向或 Y 方向测量 00 = 无操作模式 01 = X 方向测量 10 = Y 方向测量 11 = 等待中断模式	-
保留	[11:10]	保留	-
XPDATA (正常 ADC)	[9:0]	X 方向转换数值（包括正常 ADC 转换数值）数值范围：0 至 3FF	-

第 002 节_ADC 编程

编程步骤：

1. 初始化 ADC
2. 读数据，
3. 在串口上显示出来。



一，初始化 ADC 下面的函数实现对 ADC 的初始化。

```
03 void adc_init(void)
04 {
05 /* [15] : ECFLG,  1 = End of A/D conversion
06          * [14] : PRSCEN, 1 = A/D converter prescaler
enable
```

```
07          * [13:6]: PRSCVL, adc clk = PCLK / (PRSCVL +
1)
08          * [5:3] : SEL_MUX, 000 = AIN 0
09           * [2]   : STDBM
10          * [0]   : 1 = A/D conversion starts and this
bit is cleared after the startup.
11          */
12          ADCCON = (1<<14) | (49<<6) | (0<<3);
13
14          ADCDLY = 0xff;
15 }
```

- 第 12 行：配置 ADCCON 寄存器，使能 A/D 转换器预分频器，设置 A/D 转换器预分频值，上拉使能。
- 第 14 行：设置 ADC 转换启动延时值。

二，读数据 在这个读函数中启动 ADC，并且等待 ADC 转换成功。然后返回数据，

```
17 int adc_read_ain0(void)
18 {
19                /* 启动ADC */
20          ADCCON |= (1<<0);
21
22          while (!(ADCCON & (1<<15)));  /* 等待ADC 结束
*/
23
24          return ADCDAT0 & 0x3ff;
25 }
```

- 第 20 行：启动 ADC。
- 第 22 行：等待 A/D 转换结束(ADCCON 第 15 位置 1)，
- 第 24 行：返回转换的值。(ADCDAT0 寄存器的前 10 位，是保存转换后的值)。

三，ADC 测试 函数代码如下：函数功能：在串口/LCD 上打印 ADC 转换后的结果。

```
04 void adc_test(void)
05 {
06          int val;
07          double vol;
08          int m; /* 整数部分 */
```

```
09          int n; /* 小数部分 */
10
11          adc_init();
12
13          while (1)
14          {
15                  val = adc_read_ain0();
16                  vol = (double)val/1023*3.3;   /* 1023----3.3v */
17                  m = (int)vol;   /* 3.01, m = 3 */
18                  vol = vol - m;  /* 小数部分: 0.01 */
19                  n = vol * 1000;  /* 10 */
20
21                  /* 在串口上打印 */
22                  printf("vol: %d.%03dv", m, n);  /* 3.010v */
23
24                  /* 在 LCD 上打印 */
25                  //fb_print_string();
26                  }
27 }
```

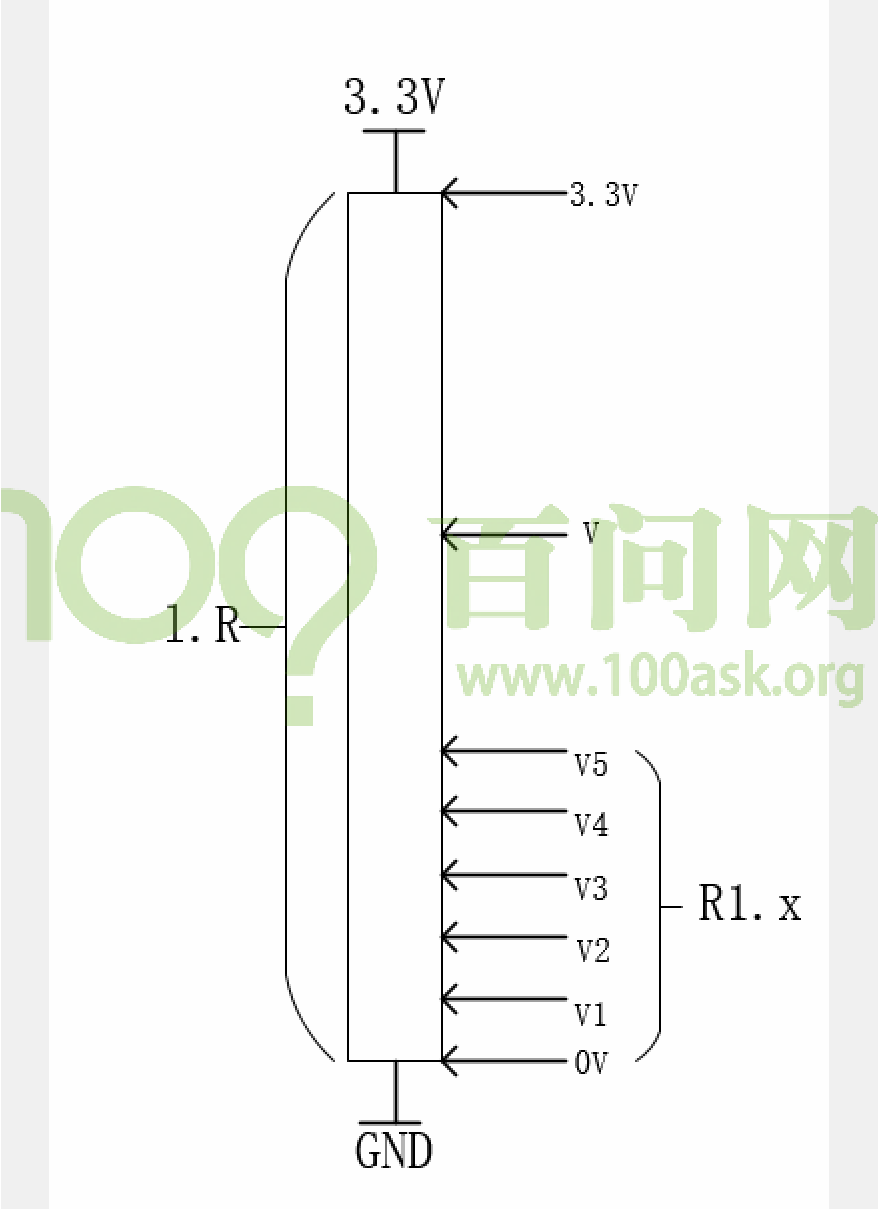
- 第 11 行：初始化 ADC.
- 第 15 行：把 ADC 转换得到的值赋值给变量 val.
- 第 16 行：把变量 val 的值转化为电压值。
- 第 17 行：取 vol 整数部分赋值给变量 m。
- 第 18 行：取 vol 的小数部分赋值给 vol。

测试 把生成的二进制文件烧录到开发板上，接上 SPI 模块，旋转可变电阻就可以在串口上看到电压值发生变化。原理图如图 1-1-2

第 003 节_电阻触摸屏硬件原

这节课我们来讲电阻触摸屏的硬件原理

假设有一个比较长的电阻 电阻是 R 上面接 3.3V 电压，下面接地



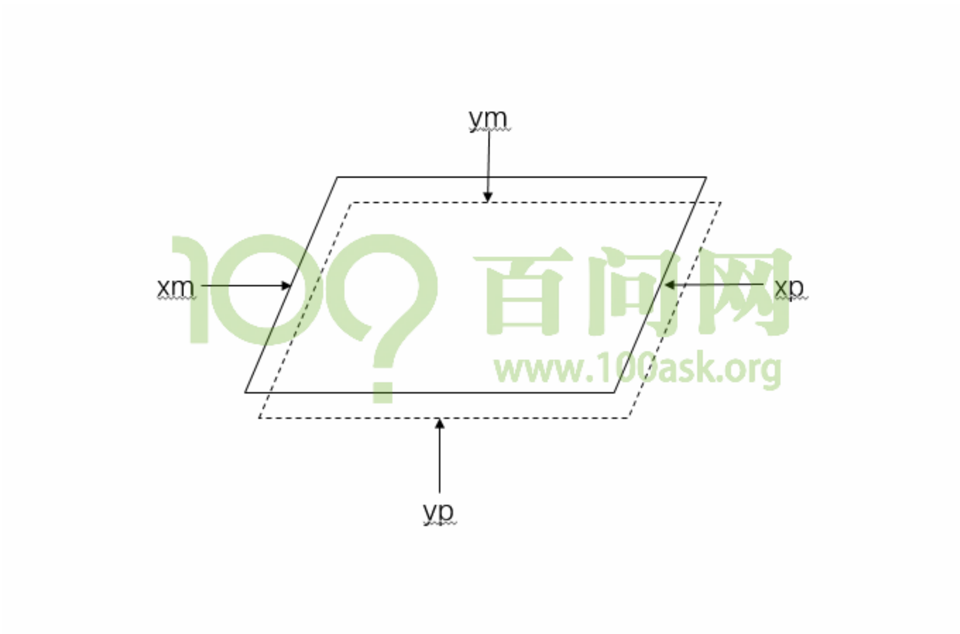
假设整个电阻的阻值是 R 某一个触电它的阻值是 R1 根据欧姆定律

3.3v/R = V/R1
V=3.3 *(R1/R)

假设 R1 是 x 坐标 R 的长度是 1 这个电阻非常的均匀，那么这个电压就等于 3.3V * (x / 1) 这个电压和这个触电的 x 坐标有一个线性关系 我使用 ADC 把这个电压算出来，就可以间接得到这个触电的 x 坐标 电阻触摸屏就是使用欧姆定律使用电阻原理作出来的

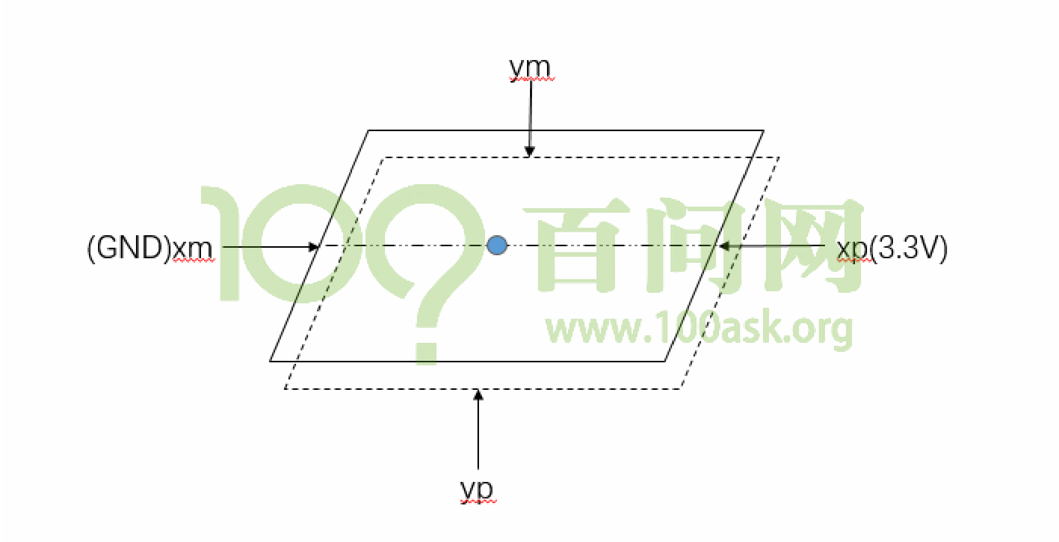
可以上百度图片搜索触摸屏，就知道了触摸屏的样子 它是一个透明的薄膜 注意 LCD 是 LCD 触摸屏是触摸屏它是两个设备 我们只不过是把触摸屏做的和 LCD 大小一样，粘在 LCD 上面 实际上触摸屏是由两层膜组成，他们靠的非常近

上面这层右边引出来，代表 xp ，p 代表正极 上面这层左边引出来，代表 xm, m 代表负极



下面这层膜 前面这条边引出来为 yp，后面这层边为 ym

假设我们手指要点击触摸屏，那么上下就会粘贴在一起，我怎么算出这个 x y 点的坐标呢？ 测量触电 x 坐标: 1 xp 接 3.3v,xm 接 GND

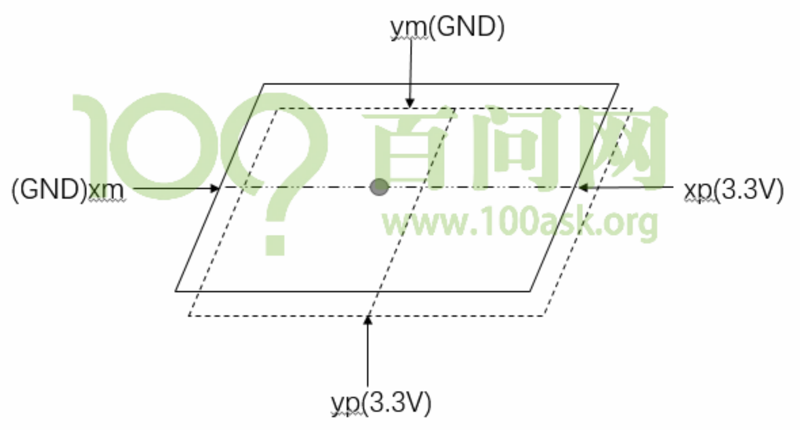


yp,ym 不接电源

2 测 yp 电压 上下膜连接在一起，我就可以通过 yp 测量这个触电的电压 这个 yp 就像探测一样，从前面的原理我们可以知道，当这个触电越靠近左边这个电压越小，越靠近右边电压越大 这个 yp 的电压就可以认为是这个触电的坐标(x 坐标)

类似的我们怎么测量触电 y 坐标 类似的 xp xm 不接电源，同样 yp 接 3.3v，ym 接 GND，这时候电流就从 yp 这里流向 ym,让后我们就可以测量 xp 电压 当按下屏幕时，上下两层膜链接在一起，这个 xp 就像探针一样，这个触电越靠近

yp 电压值越大，越靠近 ym 电压值越小



1 yp 接 3.3V ym 接 GND，xp xm 不接电源 2 测量 xp 电压,就是 y 坐标

注意 x y 坐标都是电压值，不是屏幕上 480 * 272 这些值，我们需要把电压值转换为坐标值，这需要经过一些转换

我们测量 xp yp 可以得到触点的两个方向的电压值，这些电压值和坐标是线性关系 我们现在总结下使用触摸屏的流程 1 按下触摸屏 按下触摸屏时，对于一个高效的系统，产生中断，这是触摸屏中断 2 在触摸中断程序中 启动 ADC，(获得数据，xy 坐标) 启动 ADC 就开始模数转换，不可能瞬间完成， 3 ADC 完成， 产生中断 4 ADC 中断中读取 x y 坐标，

我们来想想，在这个流程里，启动触摸屏的源头是按下触摸屏，那如果长按触摸屏，我按下之后一直不松开 滑动手指呢 那么谁来触发后续的多次 ADC 转换呢 不可能只启动一次吧 为了支持 长按 滑动操作，我们需要启用定时器

5 启动定时器 6 定时器中断发生，判断触摸屏是否仍被按下，如果按下就循环上述过程(2 在触摸中断程序中 启动 ADC，(获得数据，xy 坐标) 启动 ADC 就开始模数转换，不可能瞬间完成， 3 ADC 完成， 产生中断 4 ADC 中断中读取 x y 坐标，) 7 松开结束一个流程 这就是整个触摸屏的使用流程

在 14 章里讲解了触摸屏，他抽象了几张图

平时的时候上下两层膜并不连接，我们按下触摸屏的时候就会产生中断，那么你怎么知道产生中断，肯定是由某个引脚的电平发生变化 平时 Y_ADC/xp

是高电平 按下之后 Y_ADC 就接地了，就是被拉低了，就产生了低电平

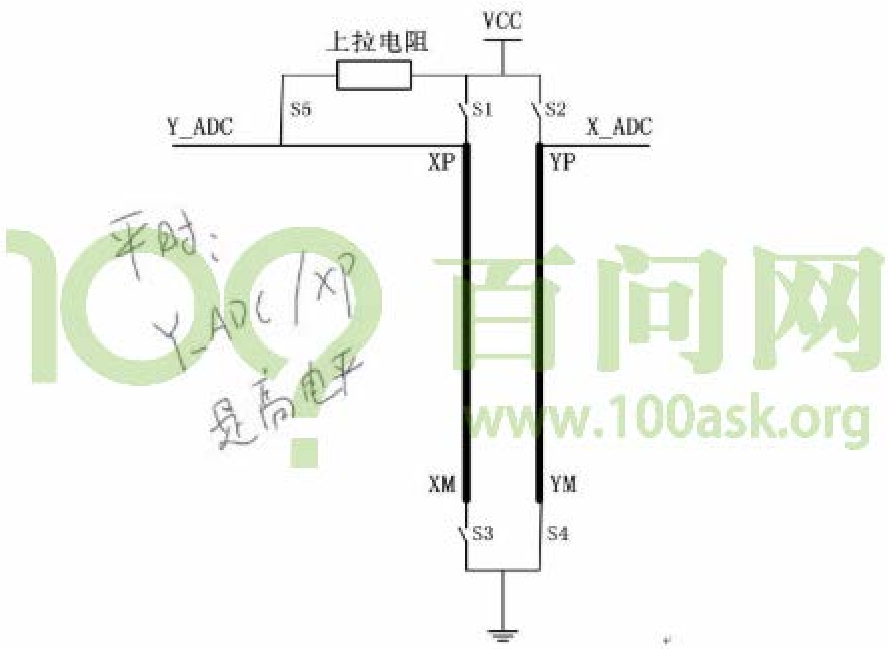


图 14.5 触摸屏处于“等待中断模式”时的等效电路

产生低电平后就知道触摸屏被按下了，这个时候就需要测量电压值 读取 x 坐标 XP XM 通电我就测量 YP 的电压，这不就是 x 点的坐标

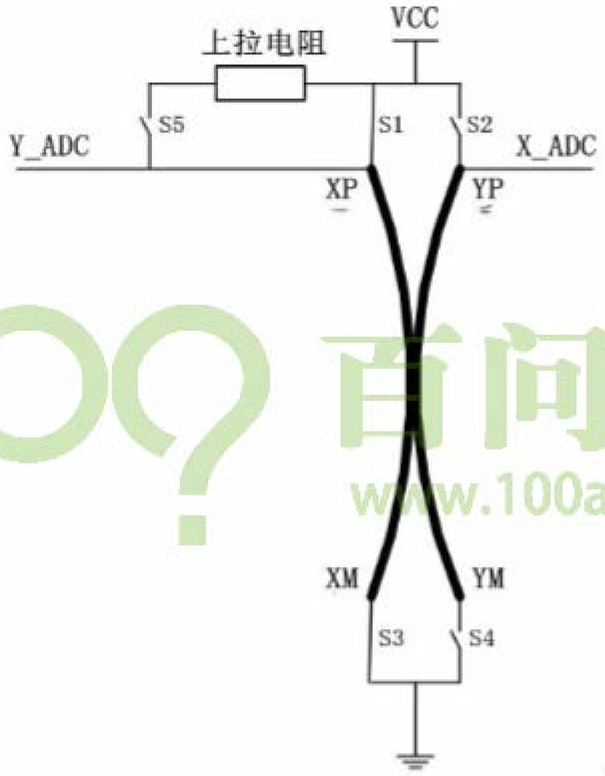


图 14.6 读取 X 坐标时的等效电路

读取 Y 坐标 YP YM 通电，按下后 XP 通电，这不就是 y 点的坐标

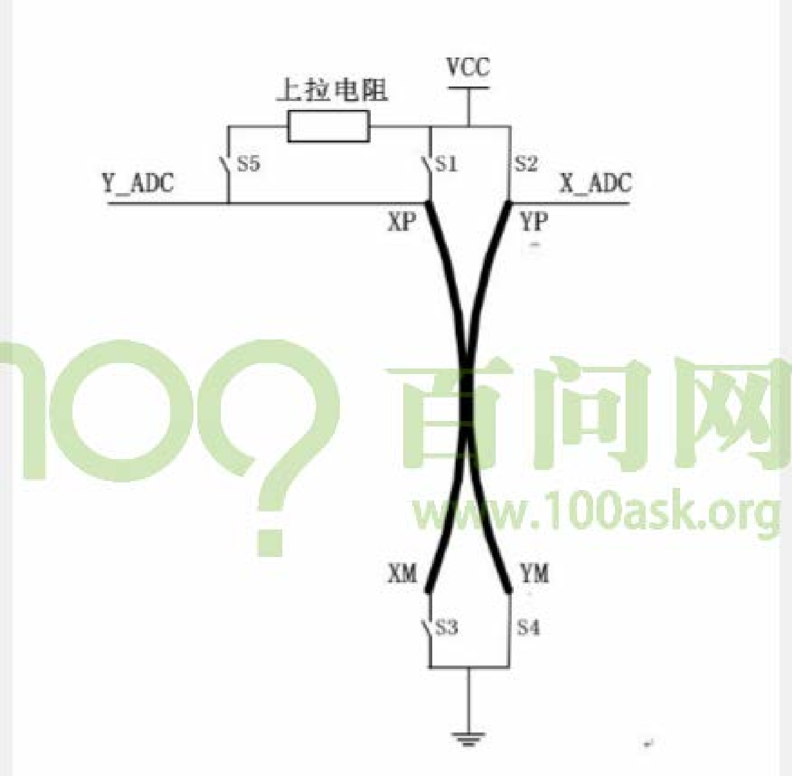


图 14.7 读取 Y 坐标时的等效电路

么

第 004 节_S3C2440 触摸屏接口

回顾上节触摸屏使用原理

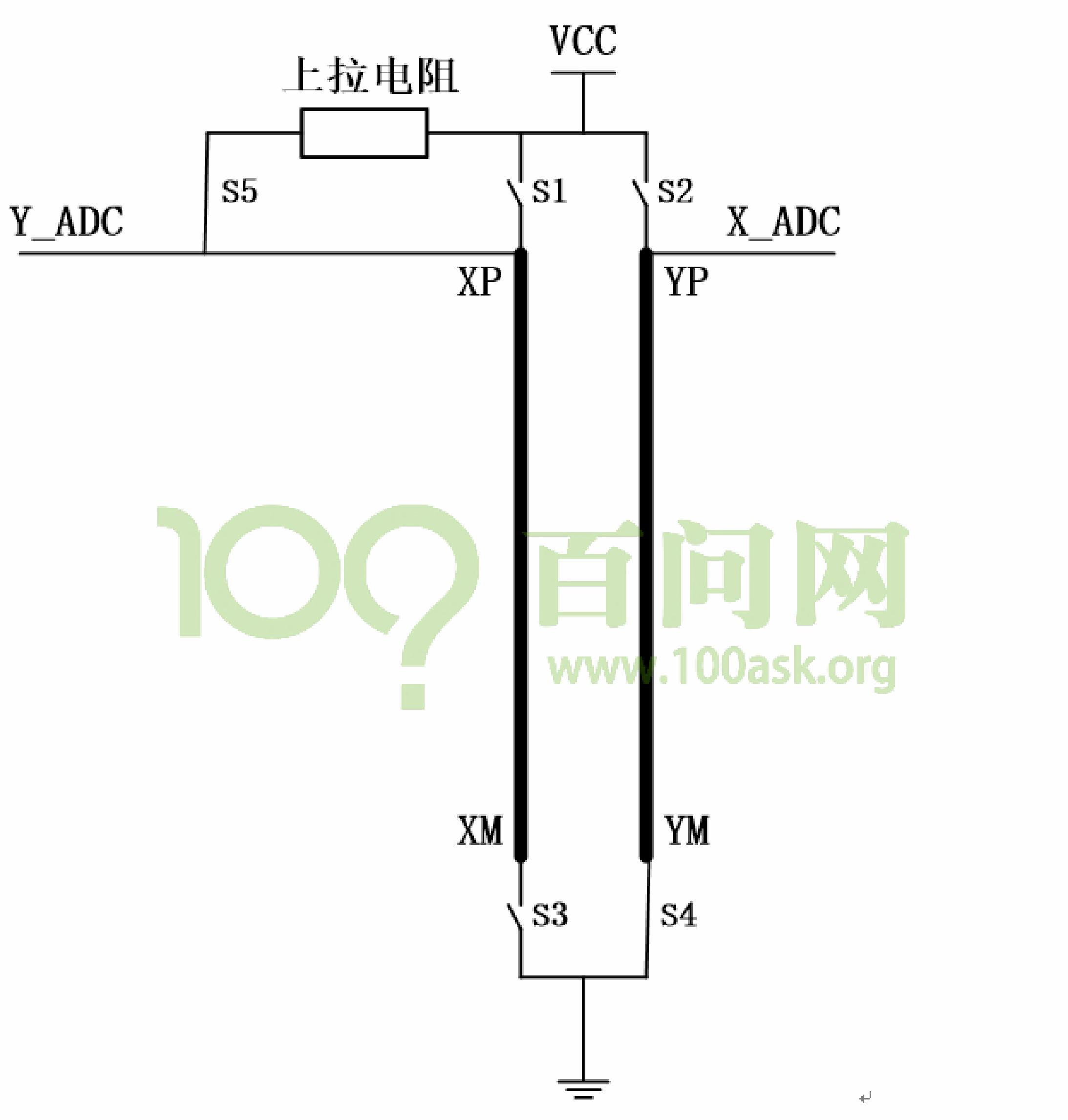


图 14.5 触摸屏处于“等待中断模式”时的等效电路

在不使用触摸屏的时候，必须要把 S1 S2 S3 断开，S4 S5 闭合，只有这样当我按下触摸屏，上面的电平才能从高变低，会产生一个中断信号

而当我去读取 X 坐标的值时

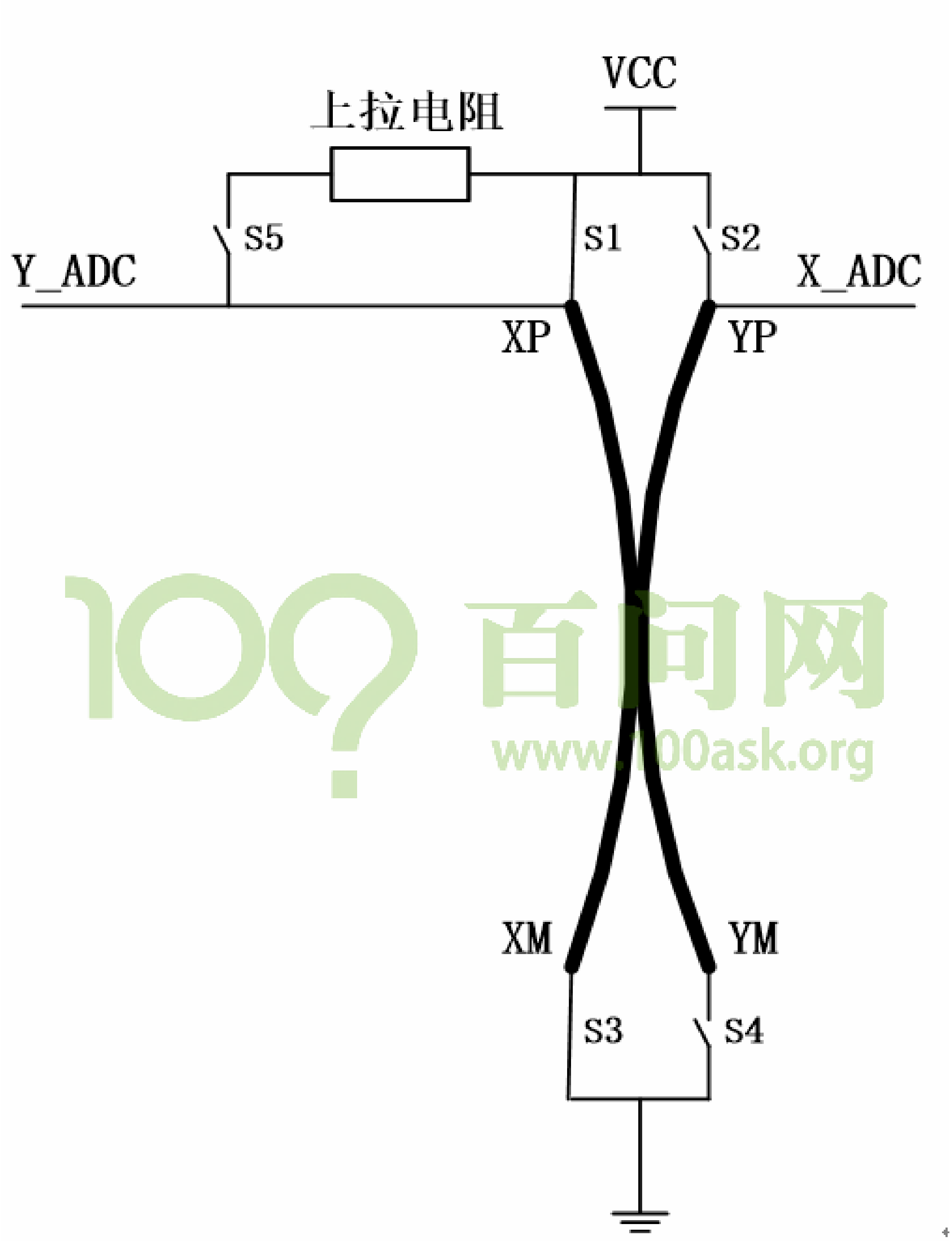


图 14.6 读取 X 坐标时的等效电路

必须让 S1 S3 闭合，这样电流才可以通过，同时让 S2 S4 S5 断开 这时候 YP 这层膜就相当于探针一样去测量电压

当我读取 y 坐标值

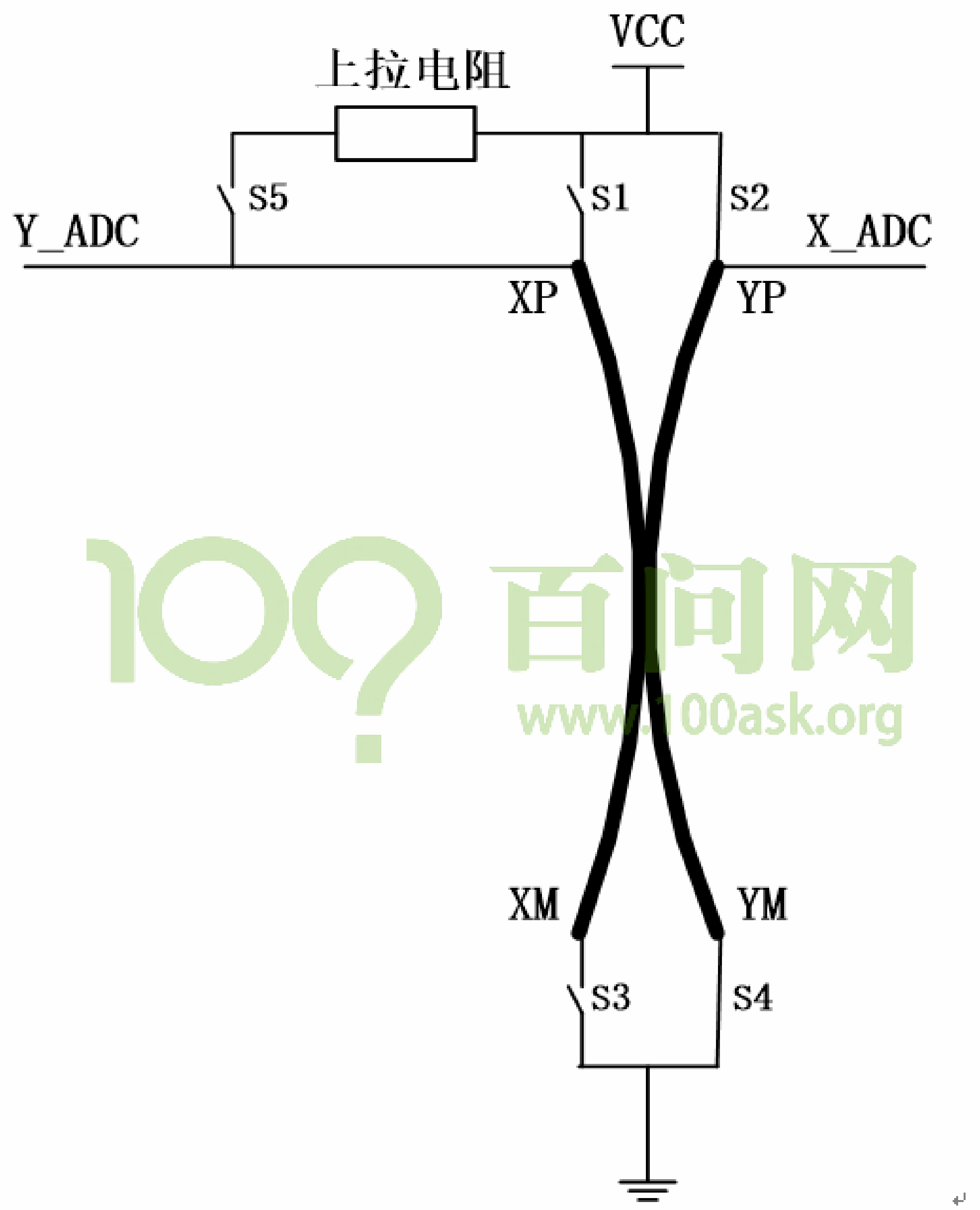


图 14.7 读取 Y 坐标时的等效电路

必须让 S2 S4 闭合，这样电流才可以流 下来，同时 S1 S3 S5 断开，这个时候 XP 这层膜就相当于探针一样，我可以来测量这里的电压，从而得到 Y 坐标的电压值

在测量 x y 坐标时，这个 S5 上拉电阻都要断开 我们需要控制这几个开关，实际上 2440 就提供了这几个开关的控制方法 打开 2440 的芯片手册看触摸屏时怎么操作的 从 440 到 450 总共 10 页不到 我们看有一个 8:1 MUX 的多路选

择器，以及 XP YP

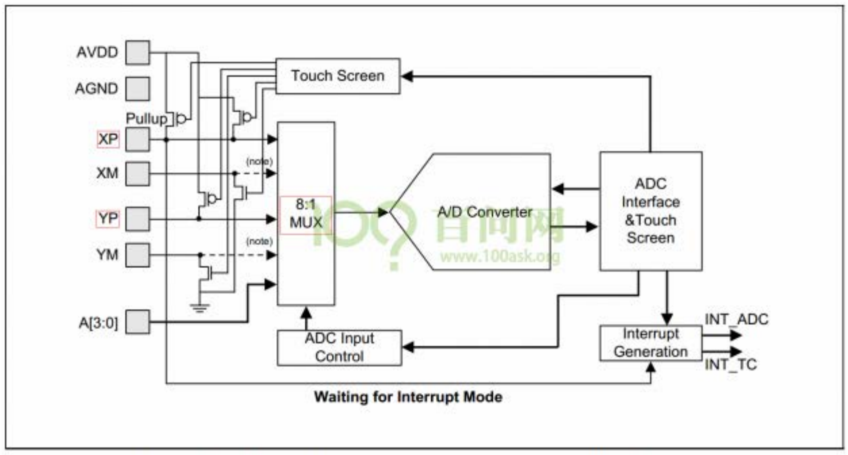


Figure 16-1. ADC and Touch Screen Interface Functional Block Diagram

442 页触摸屏接口模式

Touch Screen Interface Mode

1. Normal Conversion Mode

   Single Conversion Mode is the most likely used for General Purpose ADC Conversion. This mode can be initialized by setting the ADCCON (ADC Control Register) and completed with a read and a write to the ADCDAT0 (ADC Data Register 0).

2. Separate X/Y position conversion Mode

   Touch Screen Controller can be operated by one of two Conversion Modes. Separate X/Y Position Conversion Mode is operated as the following way. X-Position Mode writes X-Position Conversion Data to ADCDAT0, so Touch Screen Interface generates the Interrupt source to Interrupt Controller. Y-Position Mode writes Y-Position Conversion Data to ADCDAT1, so Touch Screen Interface generates the Interrupt source to Interrupt Controller.

3. Auto(Sequential) X/Y Position Conversion Mode

   Auto (Sequential) X/Y Position Conversion Mode is operated as the following. Touch Screen Controller sequentially converts X-Position and Y-Position that is touched. After Touch controller writes X-measurement data to ADCDAT0 and writes Y-measurement data to ADCDAT1, Touch Screen Interface is generating Interrupt source to Interrupt Controller in Auto Position Conversion Mode.

4. Waiting for Interrupt Mode

   Touch Screen Controller generates interrupt (INT_TC) signal when the Stylus is down. Waiting for Interrupt Mode setting value is rADCTSC=0xd3; // XP_PU, XP_Dis, XM_Dis, YP_Dis, YM_En.

   After Touch Screen Controller generates interrupt signal (INT_TC), Waiting for interrupt Mode must be cleared. (XY_PST sets to the No operation Mode)

1. 正常模式，在上节视频中我们有讲解过
2. x y 分离转换模式

看看我们的 X Y 坐标原理图，可以单独转换 X 坐标 单独转换 Y 坐标 换句话说就是逐个去测量 X Y 坐标 他首先会启动 X 坐标的 ADC 转换，转换成功后数据会保存在 ADCDAT0 里，同时会产生一个中断，在这个中断服务程序里，就可以把 X 坐标读取出来，让后可以启动 Y 坐标的转换，转换成功后数据会保存在 ADCDAT，同时会产生一个中断，进入这个中断把 Y 坐标读取出来 测量一次会产生 2 个中断，一个时 X 坐标中断，一个是 Y 坐标中断

3 自动的或连续的 X / Y 坐标转换模式 也就是说不需要单独控制，不需要单独去读取 X 坐标 Y 坐标，可以设置寄存器，让它一次性的测量 X 坐标测量 Y 坐标，X 坐标保存在 ADCDAT 0 Y 坐标保存在 ADCDAT 1 ，最后产生一个中断，也就是读取 X / Y 坐标只需要产生一次中断

4 等待中断模式 所谓等待中断模式，就是等待按下或者等待松开 对于下面这幅图，我按下的时候 XP 从高电平变为低电平，松开时，XP 从低电平变为高电平，这就是按下松开都可以检测到

我们要等待按下或者松开时　需要设置 rADCTSC =0xd3 这个值

Standby Mode 静默模式/省电模式(我们不关心这个)

Standby Mode

Standby mode is activated when ADCCON [2] is set to '1'. In this mode, A/D conversion operation is halted and ADCDAT0, ADCDAT1 register contains the previous converted data.

443 页编程要点

Programming Notes

1. The A/D converted data can be accessed by means of interrupt or polling method. With interrupt method the overall conversion time - from A/D converter start to converted data read - may be delayed because of the return time of interrupt service routine and data access time. With polling method, by checking the ADCCON[15] - end of conversion flag-bit, the read time from ADCDAT register can be determined.
2. Another way for starting A/D conversion is provided. After ADCCON[1] - A/D conversion start-by-read mode-is set to 1, A/D conversion starts simultaneously whenever converted data is read.

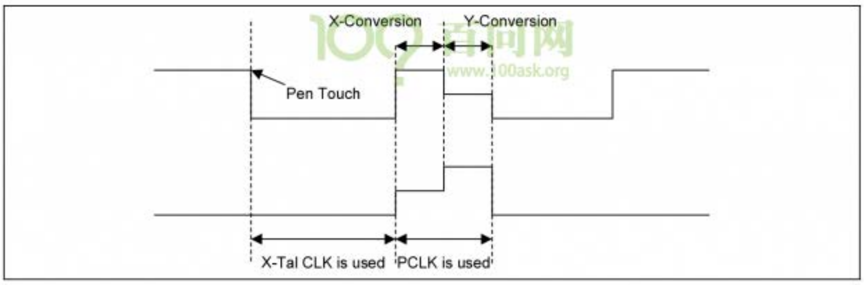


Figure 16-2. ADC and Touch Screen Operation signal

1 AD 转换数据时可以通过中断或者查询模式来得到数据，使用中断模式时，从 AD 转换开始，到得到数据可能会有些延迟，因为中断服务程序的进入和退出需要一定的时间，(也就是说，如果你对数据转换的速度要求的非常高，就可以使用查询方式)，可以查询 ADCCON[15]来判断是否转换结束

444 页 剩下就是寄存器操作

ADC AND TOUCH SCREEN INTERFACE SPECIAL REGISTERS

ADC CONTROL REGISTER (ADCCON)

Register	Address	R/W	Description	Reset Value
ADCCON	0x5800000	R/W	ADC control register	0x3FC4

ADCCON	Bit	Description	Initial State
ECFLG	[15]	End of conversion flag(Read only) 0 = A/D conversion in process 1 = End of A/D conversion	0
PRSCEN	[14]	A/D converter prescaler enable 0 = Disable 1 = Enable	0
PRSCVL	[13:6]	A/D converter prescaler value Data value: 0 ~ 255 NOTE: ADC Freqeuncy should be set less than PCLK by 5times. (Ex. PCLK=10MHZ, ADC Freq.< 2MHz)	0xFF
SEL_MUX	[5:3]	Analog input channel select 000 = AIN 0 001 = AIN 1 010 = AIN 2 011 = AIN 3 100 = YM 101 = YP 110 = XM 111 = XP	0
STDBM	[2]	Standby mode select 0 = Normal operation mode 1 = Standby mode	1
READ_ START	[1]	A/D conversion start by read 0 = Disable start by read operation 1 = Enable start by read operation	0
ENABLE_START	[0]	A/D conversion starts by enable. If READ_START is enabled, this value is not valid. 0 = No operation 1 = A/D conversion starts and this bit is cleared after the start-up.	0

ECFLG 状态位 AD 转换是否结束

PRSCEN 使能 ADC 转换 PRSCVL 设置 A/D 转换预分频值 SEL_MUX 选择输入通道，后面我们使用自动转换 XY 坐标，所以这里不需要设置 ENABLE_START 启动转换

445 页 ADCTSC 这个寄存器是重要

的

ADC TOUCH SCREEN CONTROL REGISTER (ADCTSC)

Register	Address	R/W	Description	Reset Value
ADCTSC	0x5800004	R/W	ADC Touch Screen Control Register	0x58

ADCTSC	Bit	Description	Initial State
UD_SEN	[8]	Detect Stylus Up or Down status. 0 = Detect Stylus Down Interrupt Signal. 1 = Detect Stylus Up Interrupt Signal.	0
YM_SEN	[7]	YM Switch Enable 0 = YM Output Driver Disable. 1 = YM Output Driver Enable.	0
YP_SEN	[6]	YP Switch Enable 0 = YP Output Driver Enable. 1 = YP Output Driver Disable.	1
XM_SEN	[5]	XM Switch Enable 0 = XM Output Driver Disable. 1 = XM Output Driver Enable.	0
XP_SEN	[4]	XP Switch Enable 0 = XP Output Driver Enable. 1 = XP Output Driver Disable.	1
PULL_UP	[3]	Pull-up Switch Enable 0 = XP Pull-up Enable. 1 = XP Pull-up Disable.	1
AUTO_PST	[2]	Automatically sequencing conversion of X-Position and Y-Position 0 = Normal ADC conversion. 1 = Auto Sequential measurement of X-position, Y-position.	0
XY_PST	[1:0]	Manually measurement of X-Position or Y-Position. 00 = No operation mode 01 = X-position measurement 10 = Y-position measurement 11 = Waiting for Interrupt Mode	0

UD_SEN Bit8 是用来判断触摸屏是被按下还是被松开 0 表明被按下，1 表明被松开 YM_SEN Bit7 YM 开关使能控制 S4

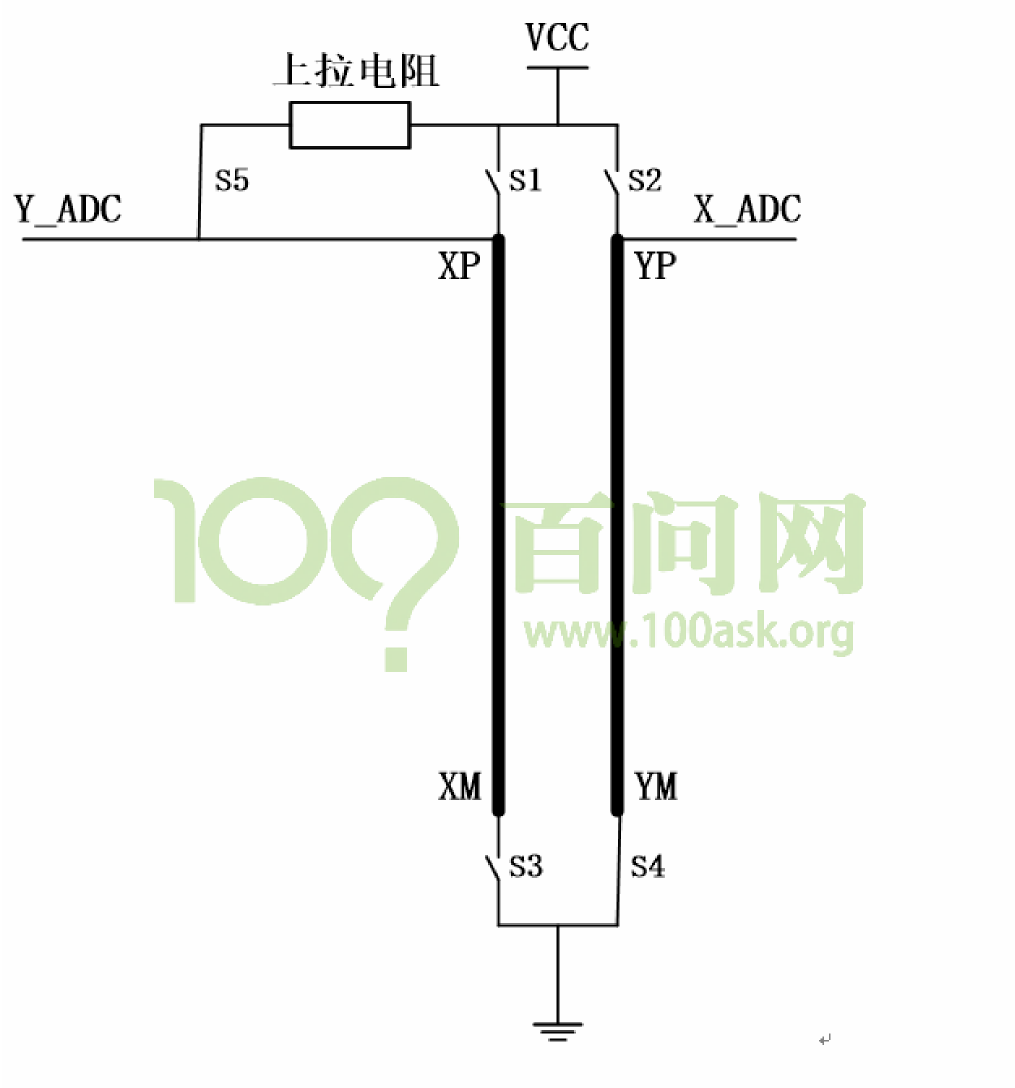


图 14.5 触摸屏处于“等待中断模式”时的等效电路

0 表示断开
1 闭合

YP_SEN Bit6 YP 开关

0 表示闭合
1 表示断开

寄存器位的含义不同
XM_SEN Bit5 XM 开关

0 断开
1 闭合

XP_SEN Bit4 XP 开关

0 闭合
1 断开

PULL_UP Bit3 控制 S5 开关

0 上拉(闭合)
1 断开

AUTO_PST Bit2 自动连续转换 X 坐标 Y 坐标

上节视频里我们设置是 0 正常的 ADC 转换 如果需要连续转换 ADC 坐标的话，需要设置为 1 ，如果需要手动转换 ADC 坐标的话，需要设置为 0

XY_PST Bit[1:0] 对于手动转换 X Y 坐标我们需要手动设置 XY_PST 里面的位，是测量 X 坐标还是测量 Y 坐标 也可以设置这两位等于 11 让其等于等待模式 也就是等待触摸屏被按下或者被松开

如果设置自动连续转换的话，Bit2 AUTO_PST 设置为 1 XY_PST 设置为 00

如果使用手动转换的话设置 AUTO_PST 为 0 XY_PST 设置为 01 手动转换 X 坐标模式 或者设置为 10 Y 坐标转换模式

447 页 ADCDATA0 ADC 数据寄存器

ADC CONVERSION DATA REGISTER (ADCDAT0)

Register	Address	R/W	Description	Reset Value
ADCDAT0	0x580000C	R	ADC conversion data register	–

ADCDAT0	Bit	Description	Initial State
UPDOWN	[15]	Up or Down state of stylus at waiting for interrupt mode. 0 = Stylus down state. 1 = Stylus up state.	–
AUTO_PST	[14]	Automatic sequencing conversion of X-position and Y-Position 0 = Normal ADC conversion. 1 = Sequencing measurement of X-position, Y-position.	–
XY_PST	[13:12]	Manually measurement of X-position or Y-position. 00 = No operation mode 01 = X-position measurement 10 = Y-position measurement 11 = Waiting for Interrupt Mode	–
Reserved	[11:10]	Reserved	–
XPDATA (Normal ADC)	[9:0]	X-Position conversion data value (include normal ADC conversion data value) Data value: 0 ~ 3FF	–

UPDOWN Bit15 可以读取这一位去判断触摸屏是按下还是松开

AUTO_PST Bit14 自动测量

XY_PST Bit[13:12] 和上面 ADCTSC 寄存器中 AUTO_PST Bit2 XY_PST Bit[1:0]原理相同

XPDATA Bit[9:0]最低 10 位用来保存 ADC 的值

448 页 ADCDAT1 寄存器 和 ADCDAT0 功能一样的，只不过保存的数据不同

ADC CONVERSION DATA REGISTER (ADCDAT1)

Register	Address	R/W	Description	Reset Value
ADCDAT1	0x5800010	R	ADC conversion data register	–

ADCDAT1	Bit	Description	Initial State
UPDOWN	[15]	Up or down state of stylus at waiting for interrupt mode. 0 = Stylus down state. 1 = Stylus up state.	–
AUTO_PST	[14]	Automatically sequencing conversion of X-position and Y-position 0 = Normal ADC conversion. 1 = Sequencing measurement of X-position, Y-position.	–
XY_PST	[13:12]	Manually measurement of X-position or Y-position. 00 = No operation mode 01 = X-position measurement 10 = Y-position measurement 11 = Waiting for interrupt mode	–
Reserved	[11:10]	Reserved	–
YPDATA	[9:0]	Y-position conversion data value Data value: 0 ~ 3FF	–

这个的低 10 位是用来保存 Y 坐标的值

接下来是 ADCUPDN 触摸屏按下或者松开检查寄存器

ADC TOUCH SCREEN UP-DOWN INT CHECK REGISTER (ADCUPDN)

Register	Address	R/W	Description	Reset Value
ADCUPDN	0x5800014	R/W	Stylus up or down interrupt status register	0x0

ADCUPDN	Bit	Description	Initial State
TSC_UP	[1]	Stylus Up Interrupt. 0 = No stylus up status. 1 = Stylus up interrupt occurred.	0
TSC_DN	[0]	Stylus Down Interrupt. 0 = No stylus down status. 1 = Stylus down interrupt occurred.	0

TST_UP Bit1 触摸屏松开中断产生

TST_DN Bit0 触摸屏按下中断产生

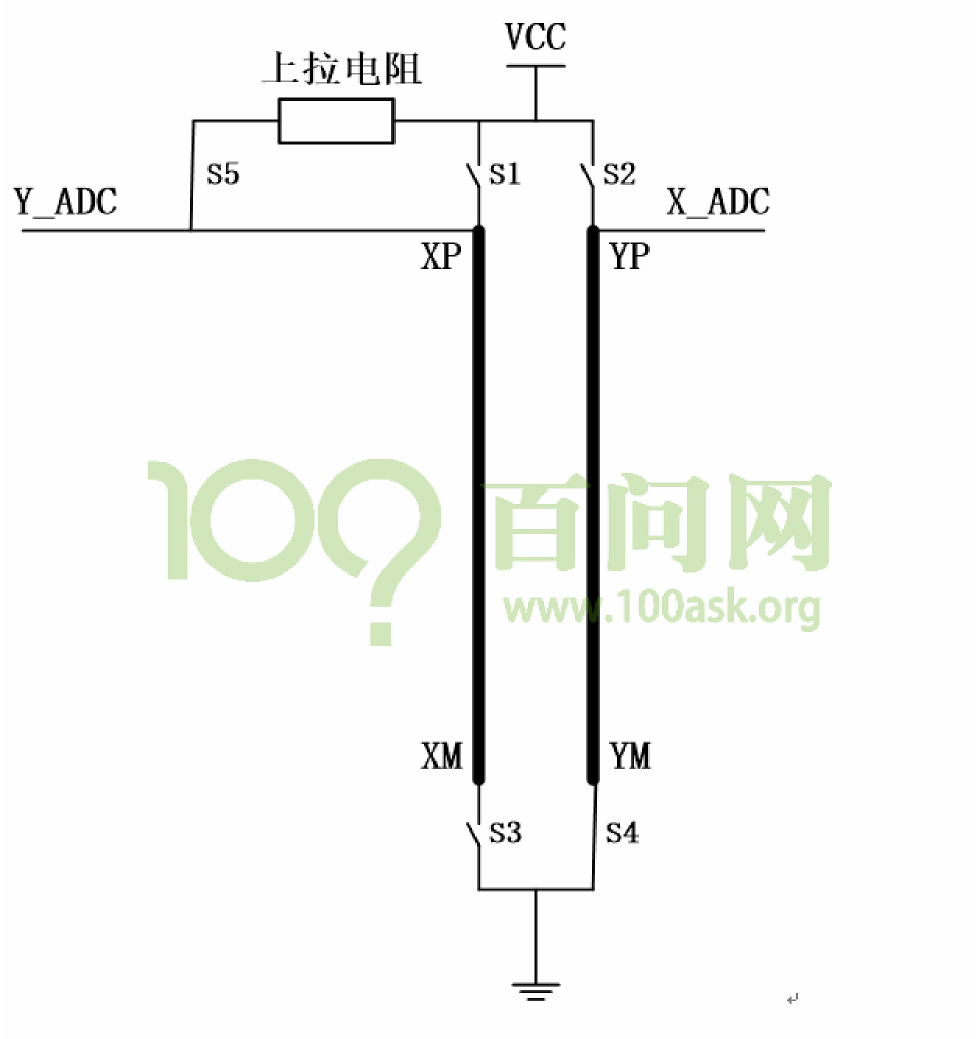
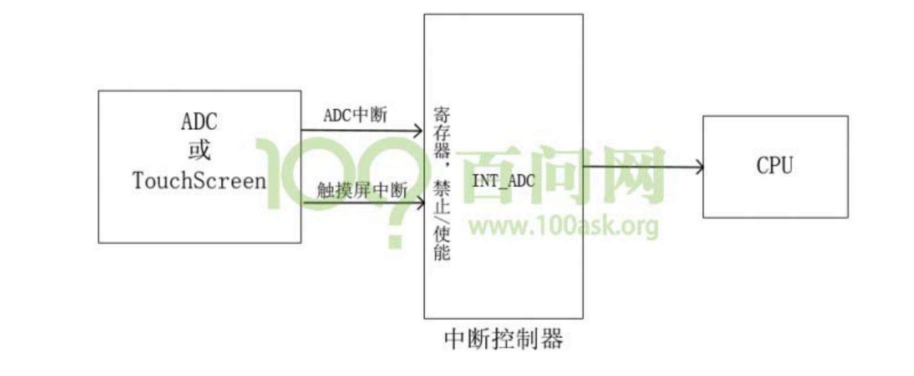


图 14.5 触摸屏处于“等待中断模式”时的等效电路

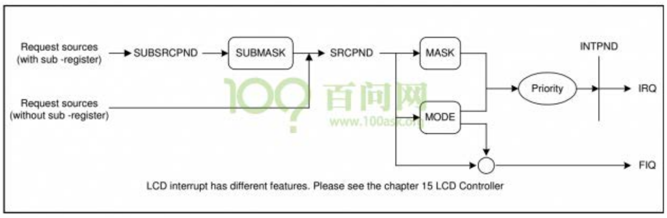
它会涉及两个中断，按下或者松开，触摸笔的状态中断，另外一个启动 ADC 以后，ADC 结束时也会产生一个中断，但是这个手册里没有看到中断的是能寄存器

那我们猜测一下，ADC 模块或者触摸屏模块一定会发出中断 首先是 ADC 或者触摸屏产生中断，通过中断控制器发送中断给 CPU



肯定有寄存器禁止 / 使能 ADC 或者触摸屏中断

我们看看中断控制器芯片手册中都需要设置什么



ADC 中断源

INTERRUPT SOURCES

The interrupt controller supports 60 interrupt sources as shown in the table below.

Sources	Descriptions	Arbiter Group
INT_ADC	ADC EOC and Touch interrupt (INT_ADC_S/INT_TC)	ARB5

ADC 结束中断或者触摸屏中断，看来他们合起来用一个中断 既然合并必然还会有一个寄存器来分辨到底是 ADC 还是触摸屏发生的中断变化

SRCPND 寄存器 31 位为 ADC 中断

Register	Address	R/W	Description	Reset Value
SRCPND	0X4A000000	R/W	Indicate the interrupt request status. 0 = The interrupt has not been requested. 1 = The interrupt source has asserted the interrupt request.	0x00000000

SOURCE PENDING (SRCPND) REGISTER (Continued)

SRCPND	Bit	Description	Initial State
INT_ADC	[31]	0 = Not requested, 1 = Requested	0

设置 Bit[31]

INTMOD 寄存器 来决定是普通中断还是快中断模式

INTERRUPT MODE (INTMOD) REGISTER

This register is composed of 32 bits each of which is related to an interrupt source. If a specific bit is set to 1, the corresponding interrupt is processed in the FIQ (fast interrupt) mode. Otherwise, it is processed in the IRQ mode (normal interrupt).

Please note that only one interrupt source can be serviced in the FIQ mode in the interrupt controller (you should use the FIQ mode only for the urgent interrupt). Thus, only one bit of INTMOD can be set to 1.

Register	Address	R/W	Description	Reset Value
INTMOD	0X4A000004	R/W	Interrupt mode register. 0 = IRQ mode 1 = FIQ mode	0x00000000

设置 Bit[31]

INTERRUPT MODE (INTMOD) REGISTER (Continued)

INTMOD	Bit	Description	Initial State
INT_ADC	[31]	0 = IRQ, 1 = FIQ	0

INTMSK 寄存器 用来表示是否屏蔽这个中断

INTERRUPT MASK (INTMSK) REGISTER

This register also has 32 bits each of which is related to an interrupt source. If a specific bit is set to 1, the CPU does not service the interrupt request from the corresponding interrupt source (note that even in such a case, the corresponding bit of SRCPND register is set to 1). If the mask bit is 0, the interrupt request can be serviced.

Register	Address	R/W	Description	Reset Value
INTMSK	0X4A000008	R/W	Determine which interrupt source is masked. The masked interrupt source will not be serviced. 0 = Interrupt service is available. 1 = Interrupt service is masked.	0xFFFFFFFF

设置 Bit[31]

INTERRUPT MASK (INTMSK) REGISTER (Continued)

INTMSK	Bit	Description	Initial State
INT_ADC	[31]	0 = Service available, 1 = Masked	1

优先级我们不需要设置

INTPND

INTERRUPT PENDING (INTPND) REGISTER

Each of the 32 bits in the interrupt pending register shows whether the corresponding interrupt request, which is unmasked and waits for the interrupt to be serviced, has the highest priority . Since the INTPND register is located after the priority logic, only one bit can be set to 1, and that interrupt request generates IRQ to CPU. In interrupt service routine for IRQ, you can read this register to determine which interrupt source is serviced among the 32 sources.

Like the SRCPND register, this register has to be cleared in the interrupt service routine after clearing the SRCPND register. We can clear a specific bit of the INTPND register by writing a data to this register. It clears only the bit positions of the INTPND register corresponding to those set to one in the data. The bit positions corresponding to those that are set to 0 in the data remains as they are.

Register	Address	R/W	Description	Reset Value
INTPND	0X4A000010	R/W	Indicate the interrupt request status. 0 = The interrupt has not been requested. 1 = The interrupt source has asserted the interrupt request.	0x00000000

设置 Bit[31]表示中断是否正在处理

INTERRUPT PENDING (INTPND) REGISTER (Continued)

INTPND	Bit	Description	Initial State
INT_ADC	[31]	0 = Not requested, 1 = Requested	0

INTOFFSET 设置 Bit[31]

INTERRUPT OFFSET (INTOFFSET) REGISTER

The value in the interrupt offset register shows which interrupt request of IRQ mode is in the INTPND register. This bit can be cleared automatically by clearing SRCPND and INTPND.

Register	Address	R/W	Description	Reset Value
INTOFFSET	0x4A000014	R	Indicate the IRQ interrupt request source	0x00000000

INT Source	The OFFSET value	INT Source	The OFFSET value
INT_ADC	31	INT_UART2	15

到底是 ADC 中断还是触摸屏中断，肯定有其他寄存器可以设置

SUBSOURCE PENDING 寄存器

INT_ADC_S Bit[10]表示 ADC 中断

INT_TC Bit[9]表示触摸屏中断

SUB SOURCE PENDING (SUBSRCPND) REGISTER

You can clear a specific bit of the SUBSRCPND register by writing a data to this register. It clears only the bit positions of the SUBSRCPND register corresponding to those set to one in the data. The bit positions corresponding to those that are set to 0 in the data remains as they are.

Register	Address	R/W	Description	Reset Value
SUBSRCPND	0X4A000018	R/W	Indicate the interrupt request status. 0 = The interrupt has not been requested. 1 = The interrupt source has asserted the interrupt request.	0x00000000

SUBSRCPND	Bit	Description	Initial State
Reserved	[31:15]	Not used	0
INT_AC97	[14]	0 = Not requested, 1 = Requested	0
INT_WDT	[13]	0 = Not requested, 1 = Requested	0
INT_CAM_P	[12]	0 = Not requested, 1 = Requested	0
INT_CAM_C	[11]	0 = Not requested, 1 = Requested	0
INT_ADC_S	[10]	0 = Not requested, 1 = Requested	0
INT_TC	[9]	0 = Not requested, 1 = Requested	0

INTSUBMSK 应该也是同样的位

INT_ADC_S Bit[10]表示 ADC 中断激活/屏蔽
INT_TC Bit[9]表示触摸屏中断激活/屏蔽

INTERRUPT SUB MASK (INTSUBMSK) REGISTER

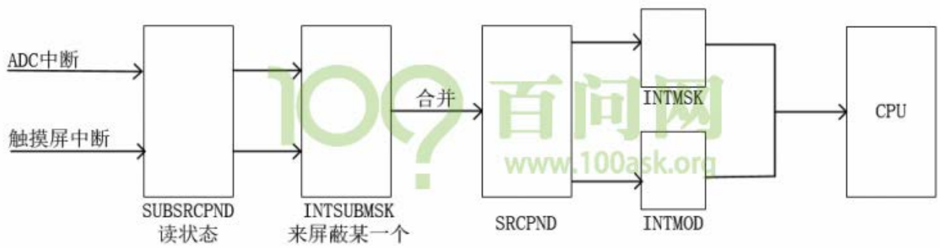
This register has 11 bits each of which is related to an interrupt source. If a specific bit is set to 1, the interrupt request from the corresponding interrupt source is not serviced by the CPU (note that even in such a case, the corresponding bit of the SUBSRCPND register is set to 1). If the mask bit is 0, the interrupt request can be serviced.

Register	Address	R/W	Description	Reset Value
INTSUBMSK	0X4A00001C	R/W	Determine which interrupt source is masked. The masked interrupt source will not be serviced. 0 = Interrupt service is available. 1 = Interrupt service is masked.	0xFFFF

INTSUBMSK	Bit	Description	Initial State
Reserved	[31:15]	Not used	0
INT_AC97	[14]	0 = Service available, 1 = Masked	1
INT_WDT	[13]	0 = Service available, 1 = Masked	1
INT_CAM_P	[12]	0 = Service available, 1 = Masked	1
INT_CAM_C	[11]	0 = Service available, 1 = Masked	1
INT_ADC_S	[10]	0 = Service available, 1 = Masked	1
INT_TC	[9]	0 = Service available, 1 = Masked	1

我们可以通过 INTSUBMSK 来屏蔽 ADC 中断或者 TouchScreen 中断 当然也可以是能某个中断 可以通过 SUBSRCPND 来分辨到底产生那个中断 INTSUBMSK 和 SUBSOURCPND 这两个寄存器都会汇集到一起 变成一个叫做 INT_ADC 的中断来发送给 CPU

框图就是这样



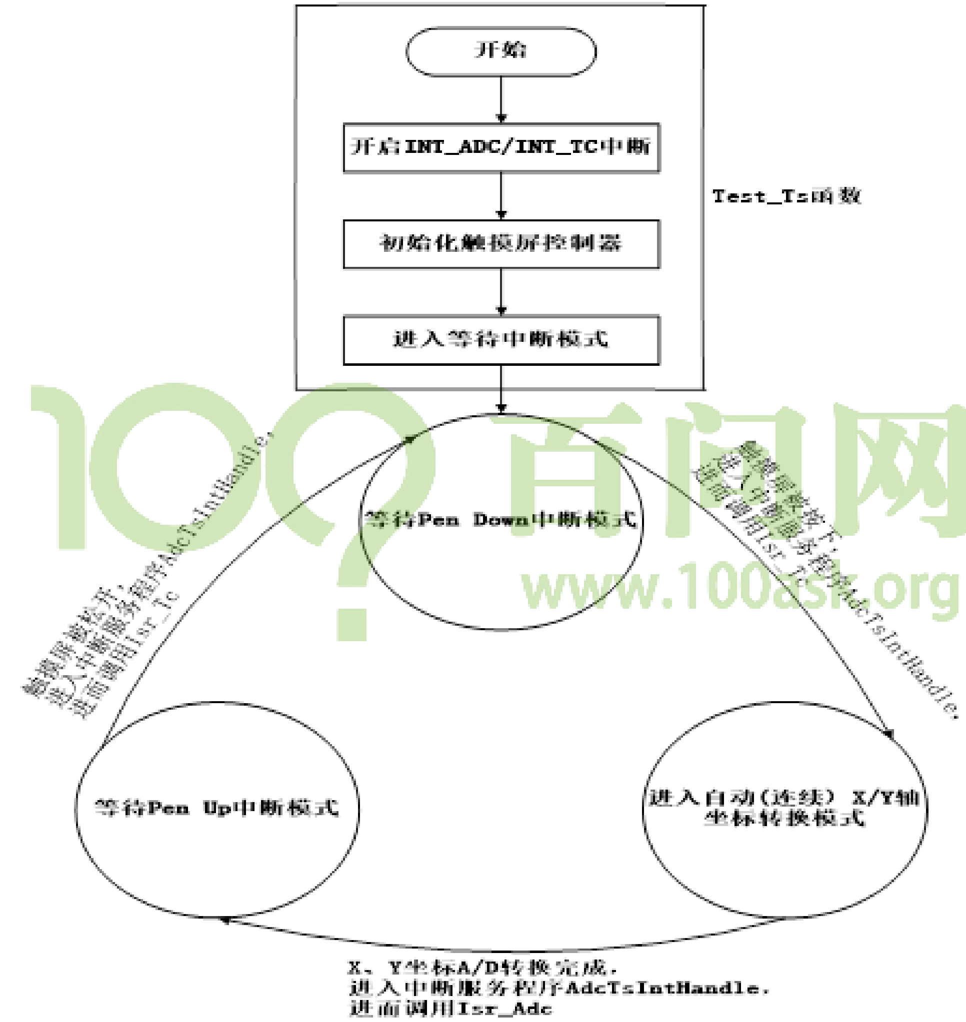
我们怎么写程序？ 写出一个框架

1 初始化 ADC/TouchScreen 接口
ADCCON 时钟接口
2 一开始触摸屏是没有被按下的，设置 TS 处于等待中断模式
3 设置中断
INTSUBMSK 使能 ADC 中断和触摸屏中断
还有 INTMSK 设置这个寄存器使能 ANT_ADC 让他能够发给 CPU
4 按下触摸屏，进入 TS 中断
    4.1 进入自动采集模式(自动转换 XY 坐标)
    4.2 启动 ADC
5.转换完之后产生 ADC 中断
    5.1 读数据
    5.2 再次进入 “等待中断”模式
    5.3 启动定时器，处理长按或者滑动
6 定时器中断
    6.1 判断是否松开，若松开结束

6.2 若按下重新执行 4.2 启动 ADC 步骤

第 005 节_触摸屏编程_按下松开检测

- 开始触摸屏编程，关于触摸屏编程大概会分为 3 个小节
- 第 006 节_触摸屏编程_ADC 中断
- 第 007 节_触摸屏编程_定时器程序优化
- 参考《嵌入式 Linux 应用开发完全手册》第 14 章 ADC 和触摸屏接口
- 可以参考下面这张图



看懂这张图的关键点在于 里面有个中断程序 AdcTsIntHandle 它是总的中断，这里面要分辨 if 如果是 ADC 中断 那么就调用 Isr_adc 来处理中段 else if 如果是触摸屏中断，那么就调用 Isr_tc 中断 这些都是总中断具体的中断

我们看看是怎么做的

- 一开始设置中断
- 初始化触摸屏控制器，进入等待中断模式
- 这个时候如果按下触摸屏就会进入 Pen Down 中断
- 就会进入 AdcTsIntHandle 这个总中断函数
- 这里面分辨是按下触摸屏
- 进入自动(连续) X/Y 轴坐标转换模式，启动 ADC,
- ADC 结束之后会产生一个 ADC 中断
- 又再次进入这个 AdcTsIntHandle 总中断
- 这里面分辨是 ADC 中断，这里面调用 Isr_Adc
- 我可以读出这里面的数据，再次设置寄存器
- 进入等待 Pen UP 中断模式
- 松开触摸笔会再次产生一个中断
- 进入总中断 AdcTsIntHandle 这里面分辨，原来是松开了触摸笔，再次调用 Isr_tc

- 这里面又会设置进入等待 Pen Down 中断模式
-

我们开始写代码，再上一个视频 ADC 代码上进行修改

002_touchscreen_018_005/adc_touchscreen

我们在 adc_touchscreen 目录下添加几个文件

touchscreen_test.c
touchscreen.c

我们打开 touchscreen.c 文件

```
void touchscreen_init(void)
{
    看看上面流程图
    /*1 设置触摸屏接口:也就是寄存器 */


    /*2 设置中断我们需要提供中断处理函数 */


    /*3 让触摸屏控制器进入"等待中断模式" */

}
```

我们设置中断处理函数

```
void AdcTsIntHandle(void)
{
}
```

看一下之前我们是怎么写中断的,看一下 interrupt.c 文件

```
void key_eint_irq(int irq)
```

有个中断号

那么我们也定义个 int irq 参数

```
void AdcTsIntHandle(int irq)
```

我们在这个里面分辨一下

```
if (SUBSRCPND & (1<<TC_INT_BIT))  /* 如果是触摸屏中断 */

/*调用*/
```

```
    Isr_Tc();

else if //如果是 ADC 中断

//调用
    Isr_Adc();
```
我们等会实现这两个函数

我们继续写代码
```
void touchscreen_init(void)
{
    看看上面流程图

    /*1 设置中断我们需要提供中断处理函数 */
    adc_ts_int_init();

    /*2 设置触摸屏接口:也就是寄存器 */
    adc_ts_reg_init();

    /*3 让触摸屏控制器进入"等待中断模式" */
    enter_wait_pen_down_mode();

}
```

我们先来实现 adc_ts_int_init

```
void adc_ts_int_init(void)
{
    /*注册中断处理函数*/

//怎么注册看之前的代码
    register_irq(irq, irq_handle);
```

Register	Address	R/W	Description	Reset Value
SRCPND	0X4A000000	R/W	Indicate the interrupt request status. 0 = The interrupt has not been requested. 1 = The interrupt source has asserted the interrupt request.	0x00000000

SOURCE PENDING (SRCPND) REGISTER (Continued)

SRCPND	Bit	Description	Initial State
INT_ADC	[31]	0 = Not requested, 1 = Requested	0

我们是 31 号中断

register_irq(31, AdcTsIntHandle);

/*使能中断*/ 怎么使能中断

我们需要把 INTSUBMISK 寄存器的 Bit9 Bit10 设置为 0 宏定义

```
#define ADC_INT_BIT (10)
#define TC_INT_BIT  (9)
```

使能中断，清零

```
INTSUBMSK &= ~((1<<ADC_INT_BIT) | (1<<TC_INT_BIT));
```

还有 INTMSK 我们也需要把 Bit31 清零

```
#define INT_ADC_TC (31)
```

Bit31 位清零操作

```
INTMSK    &= ~(1<<INT_ADC_TC);
```

这句可以不用设置，因为 register_irq 已经设置

假设产生中断就会进入 AdcTsIntHandle 函数中 分辨是触摸屏终端还是 ADC 中断

```
void AdcTsIntHandle(int irq)
{
    if  /* 如果是触摸屏中断 */
            Isr_Tc();

    if  /* ADC 中断 */
```

```
        Isr_Adc();
}
```

如何进行分辨

SUB SOURCE PENDING (SUBSRCPND) REGISTER

You can clear a specific bit of the SUBSRCPND register by writing a data to this register. It clears only the bit positions of the SUBSRCPND register corresponding to those set to one in the data. The bit positions corresponding to those that are set to 0 in the data remains as they are.

Register	Address	R/W	Description	Reset Value
SUBSRCPND	0X4A000018	R/W	Indicate the interrupt request status. 0 = The interrupt has not been requested. 1 = The interrupt source has asserted the interrupt request.	0x00000000

SUBSRCPND	Bit	Description	Initial State
Reserved	[31:15]	Not used	0
INT_AC97	[14]	0 = Not requested, 1 = Requested	0
INT_WDT	[13]	0 = Not requested, 1 = Requested	0
INT_CAM_P	[12]	0 = Not requested, 1 = Requested	0
INT_CAM_C	[11]	0 = Not requested, 1 = Requested	0
INT_ADC_S	[10]	0 = Not requested, 1 = Requested	0
INT_TC	[9]	0 = Not requested, 1 = Requested	0

```
if (SUBSRCPND & (1<<TC_INT_BIT))  /* 如果是触摸屏中断 */
    Isr_Tc();
if (SUBSRCPND & (1<<ADC_INT_BIT))  /* ADC 中断 */
    Isr_Adc();
```

```
//我们要引用寄存器地址头文件
#include "../s3c2440_soc.h"
#define ADC_INT_BIT (10)
#define TC_INT_BIT  (9)
#define INT_ADC_TC   (31)
/* ADCTSC's bits */
#define WAIT_PEN_DOWN    (0<<8)
#define WAIT_PEN_UP      (1<<8)
#define YM_ENABLE        (1<<7)
#define YM_DISABLE       (0<<7)
#define YP_ENABLE        (0<<6)
#define YP_DISABLE       (1<<6)

#define XM_ENABLE        (1<<5)
#define XM_DISABLE       (0<<5)

#define XP_ENABLE        (0<<4)
#define XP_DISABLE       (1<<4)

#define PULLUP_ENABLE    (0<<3)
#define PULLUP_DISABLE   (1<<3)

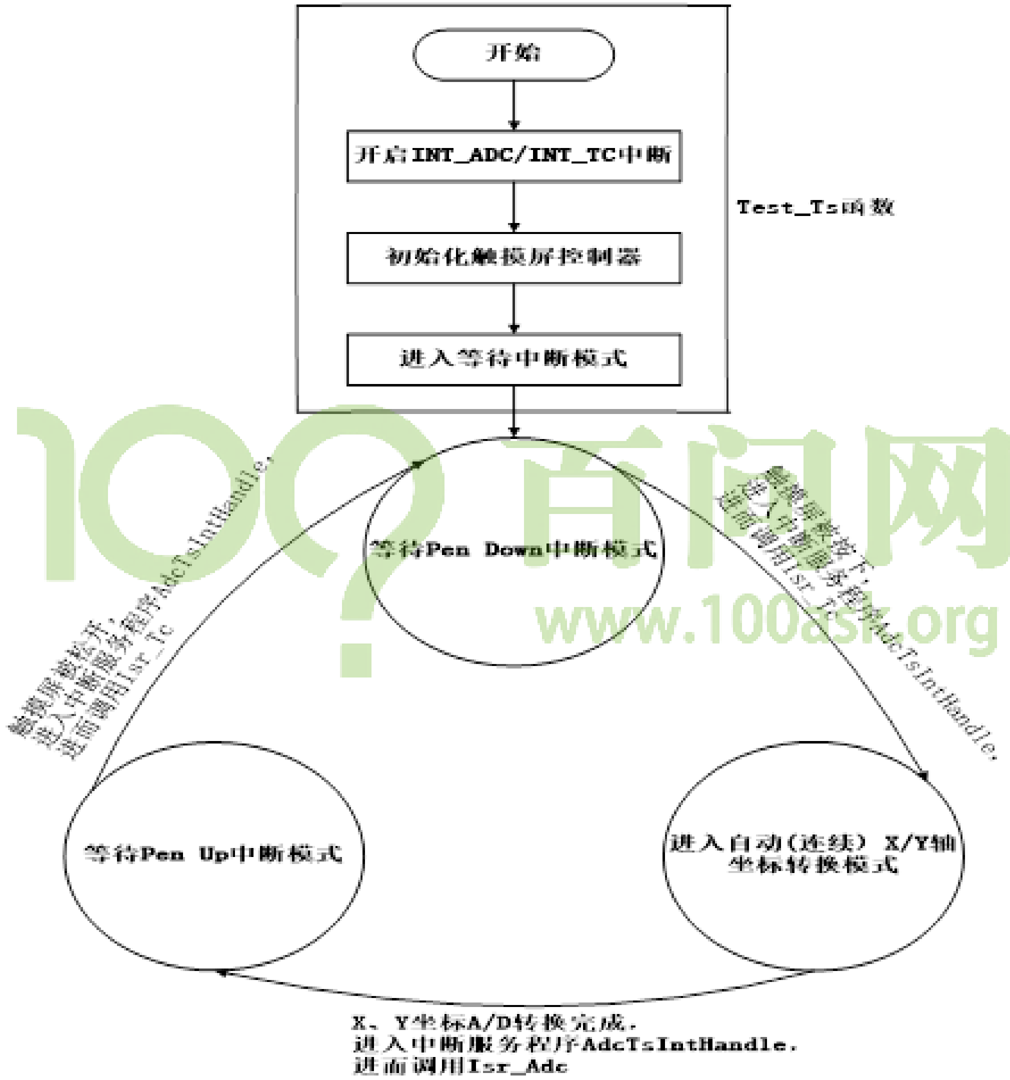
#define AUTO_PST         (1<<2)
```

```
#define WAIT_INT_MODE    (3)
#define NO_OPR_MODE     (0)

void enter_wait_pen_down_mode(void)
{
    ADCTSC = WAIT_PEN_DOWN | PULLUP_ENABLE | YM_ENABLE |
YP_DISABLE | XP_DISABLE | XM_DISABLE | WAIT_INT_MODE;
}

void enter_wait_pen_up_mode(void)
{
    ADCTSC = WAIT_PEN_UP | PULLUP_ENABLE | YM_ENABLE |
YP_DISABLE | XP_DISABLE | XM_DISABLE | WAIT_INT_MODE;
}

//读一下寄存器 找到触摸屏的寄存器 触摸笔按下松开状态寄存器
```



我们可以读它

Bit1 表示 up
Bit0 表示 down

```
void Isr_Tc(void)
{
    printf("ADCUPDN = 0x%x, ADCDAT0 = 0x%x, ADCDAT1 = 0x%x, ADCTSC = 0x%x\n\r", ADCUPDN, ADCDAT0, ADCDAT1, ADCTSC);

    if (ADCDAT0 & (1<<15))
    {
        printf("pen up\n\r");
        enter_wait_pen_down_mode();
    }
```

```
        else
        {
                printf("pen down\n\r");

                /* 进入"等待触摸笔松开的模式" */
                enter_wait_pen_up_mode();
        }
    }

    void AdcTsIntHandle(int irq)
    {
        if (SUBSRCPND & (1<<TC_INT_BIT))  /* 如果是触摸屏中断
*/
                Isr_Tc();

    // if (SUBSRCPND & (1<<ADC_INT_BIT))  /* ADC 中断 */
    //          Isr_Adc();
        SUBSRCPND = (1<<TC_INT_BIT) | (1<<ADC_INT_BIT);
    }

    void adc_ts_int_init(void)
    {
        SUBSRCPND = (1<<TC_INT_BIT) | (1<<ADC_INT_BIT);

        /* 注册中断处理函数 */
        register_irq(31, AdcTsIntHandle);

        /* 使能中断 */
        INTSUBMSK &= ~((1<<ADC_INT_BIT) | (1<<TC_INT_BIT));
        //INTMSK    &= ~(1<<INT_ADC_TC);
    }


    void adc_ts_reg_init(void)
    {
        /* [15] : ECFLG,  1 = End of A/D conversion
         * [14] : PRSCEN, 1 = A/D converter prescaler
enable
         * [13:6]: PRSCVL, adc clk = PCLK / (PRSCVL + 1)
```

```
         * [5:3] : SEL_MUX, 000 = AIN 0
         * [2]   : STDBM
         * [0]   : 1 = A/D conversion starts and this bit
is cleared after the startup.
         */
        ADCCON = (1<<14) | (49<<6) | (0<<3);

        ADCDLY = 0xff;
    }


    void touchscreen_init(void)
    {
        /* 设置触摸屏接口:寄存器 */
        adc_ts_reg_init();

        printf("ADCUPDN = 0x%x, SUBSRCPND = 0x%x, SRCPND =
0x%x\n\r", ADCUPDN, SUBSRCPND, SRCPND);


        /* 设置中断 */
        adc_ts_int_init();


        /* 让触摸屏控制器进入"等待中断模式" */
        enter_wait_pen_down_mode();
    }
```

第 006 节_触摸屏编程_ADC 中断

这节课我们加上 ADC 中断把触点的 xy 坐标读出来
查看 touchscreen.c

写出这个自动测量的函数

```
void enter_auto_measure_mode(void)
{
//现在是自动测量，我们没有机会分别设置这些开关
```

设置 AUTO_PST =1

XY_PST = 00

```
    ADCTSC = AUTO_PST | NO_OPR_MODE;
```

}

//现在是自动测量，我们没有机会分别设置这些开关

ADC TOUCH SCREEN CONTROL REGISTER (ADCTSC)

Register	Address	R/W	Description	Reset Value
ADCTSC	0x5800004	R/W	ADC Touch Screen Control Register	0x58

ADCTSC	Bit	Description	Initial State
UD_SEN	[8]	Detect Stylus Up or Down status. 0 = Detect Stylus Down Interrupt Signal. 1 = Detect Stylus Up Interrupt Signal.	0
YM_SEN	[7]	YM Switch Enable 0 = YM Output Driver Disable. 1 = YM Output Driver Enable.	0
YP_SEN	[6]	YP Switch Enable 0 = YP Output Driver Enable. 1 = YP Output Driver Disable.	1
XM_SEN	[5]	XM Switch Enable 0 = XM Output Driver Disable. 1 = XM Output Driver Enable.	0
XP_SEN	[4]	XP Switch Enable 0 = XP Output Driver Enable. 1 = XP Output Driver Disable.	1
PULL_UP	[3]	Pull-up Switch Enable 0 = XP Pull-up Enable. 1 = XP Pull-up Disable.	1
AUTO_PST	[2]	Automatically sequencing conversion of X-Position and Y-Position 0 = Normal ADC conversion. 1 = Auto Sequential measurement of X-position, Y-position.	0
XY_PST	[1:0]	Manually measurement of X-Position or Y-Position. 00 = No operation mode 01 = X-position measurement 10 = Y-position measurement 11 = Waiting for Interrupt Mode	0

进入中断处理函数

```
void AdcTsIntHandle(int irq)
{
    if (SUBSRCPND & (1<<TC_INT_BIT))  /* 如果是触摸屏中断 */
            Isr_Tc();

    if (SUBSRCPND & (1<<ADC_INT_BIT))  /* ADC 中断，则会进入 Adc 中断处理函数 */

            Isr_Adc();
    SUBSRCPND = (1<<TC_INT_BIT) | (1<<ADC_INT_BIT);
}
```

进入触摸屏中断处理函数

```
void Isr_Tc(void)
```

```
    {
        //printf("ADCUPDN = 0x%x, ADCDAT0 = 0x%x, ADCDAT1 = 
0x%x, ADCTSC = 0x%x\n\r", ADCUPDN, ADCDAT0, ADCDAT1, 
ADCTSC);

        if (ADCDAT0 & (1<<15))
        {
            //printf("pen up\n\r");
            enter_wait_pen_down_mode();
        }
        else
        {
            /* 进入"自动测量"模式 */
            enter_auto_measure_mode();

            /* 启动ADC */


    ENABLE_START = 1 就可以了
            ADCCON |= (1<<0);
        }
    }
```

/* 启动 ADC */

ADC CONTROL REGISTER (ADCCON)

Register	Address	R/W	Description	Reset Value
ADCCON	0x5800000	R/W	ADC control register	0x3FC4

ADCCON	Bit	Description	Initial State
ECFLG	[15]	End of conversion flag(Read only) 0 = A/D conversion in process 1 = End of A/D conversion	0
PRSCEN	[14]	A/D converter prescaler enable 0 = Disable 1 = Enable	0
PRSCVL	[13:6]	A/D converter prescaler value Data value: 0 ~ 255 NOTE: ADC Freqeuncy should be set less than PCLK by 5times. (Ex. PCLK=10MHZ, ADC Freq.< 2MHz)	0xFF
SEL_MUX	[5:3]	Analog input channel select 000 = AIN 0 001 = AIN 1 010 = AIN 2 011 = AIN 3 100 = YM 101 = YP 110 = XM 111 = XP	0
STDBM	[2]	Standby mode select 0 = Normal operation mode 1 = Standby mode	1
READ_ START	[1]	A/D conversion start by read 0 = Disable start by read operation 1 = Enable start by read operation	0
ENABLE_START	[0]	A/D conversion starts by enable. If READ_START is enabled, this value is not valid. 0 = No operation 1 = A/D conversion starts and this bit is cleared after the start-up.	0

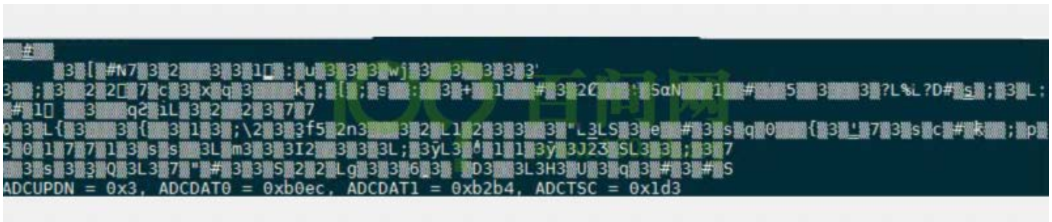
Adc 中断处理函数

```
void Isr_Adc(void)
{
    进入 adc 中断后，等待触摸笔松开
    int x = ADCDAT0 & 0x3ff;
    int y = ADCDAT1 & 0x3ff;

    printf("x = %08d, y = %08d\n\r", x, y);

//等待触摸笔松开模式
    enter_wait_pen_up_mode();
}
```

烧写 实验发现打印一堆乱码



应该是 printf 函数出了问题 打开 my_printf.c 文件，找到 printf 函数应该是处理第二个数据的时候，没有设置初始值

```
    /*reference :   int vprintf(const char *format, 
va_list ap); */
    static int my_vprintf(const char *fmt, va_list ap)
    {
        char lead=' ';
        int  maxwidth=0;

         for(; *fmt != '\0'; fmt++)
         {
                    if (*fmt != '%') {
                            outc(*fmt);
                            continue;
                    }
        //碰到 % 就重新处理，  初始值应该重新设置初始值上去
                lead=' ';
                maxwidth=0;

                //format : %08d, %8d,%d,%u,%x,%f,%c,%s
                    fmt++;
                if(*fmt == '0'){
                        lead = '0';
                        fmt++;
                }

                while(*fmt >= '0' && *fmt <= '9'){
                        maxwidth *=10;
                        maxwidth += (*fmt - '0');
                        fmt++;
                }

                        switch (*fmt) {
                case 'd': out_num(va_arg(ap, int),
10,lead,maxwidth); break;
                case 'o': out_num(va_arg(ap, unsigned int),
8,lead,maxwidth); break;
                case 'u': out_num(va_arg(ap, unsigned int),
10,lead,maxwidth); break;
                case 'x': out_num(va_arg(ap, unsigned int),
16,lead,maxwidth); break;
                        case 'c': outc(va_arg(ap, int   ));
break;
                        case 's': outs(va_arg(ap, char *));
break;
```

```
                    default:
                            outc(*fmt);
                            break;
                    }
    }
    return 0;
}
```

```
ADCUPDN = 0x0, SUBSRCPND = 0x2, SRCPND = 0x2000000
x = 0x00000006, y = 0x000003dc
x = 0x0000014d, y = 0x000003c0
x = 0x00000167, y = 0x000003a0
x = 0x000000bf, y = 0x000003bb
x = 0x0000012a, y = 0x00000394
x = 0x000000e7, y = 0x000003a4
x = 0x000000c9, y = 0x000003b1
x = 0x000000f2, y = 0x0000039e
x = 0x000000f2, y = 0x0000038c
x = 0x000000ee, y = 0x00000382
x = 0x000000f7, y = 0x00000365
x = 0x000000fa, y = 0x00000357
x = 0x000000d1, y = 0x0000035f
x = 0x000000c2, y = 0x00000367
x = 0x000000df, y = 0x0000033e
x = 0x000000a4, y = 0x0000035d
x = 0x000000ab, y = 0x00000352
x = 0x000000c2, y = 0x0000033a
x = 0x0000005b, y = 0x00000391
x = 0x000000c9, y = 0x000002f0
x = 0x00000106, y = 0x000002db
x = 0x000000f6, y = 0x000002d0
x = 0x00000022, y = 0x000003cf
x = 0x0000008c, y = 0x0000034e
x = 0x00000078, y = 0x0000036e
x = 0x0000009a, y = 0x0000032c
x = 0x0000006c, y = 0x00000342
x = 0x0000008c, y = 0x0000032d
x = 0x00000070, y = 0x00000347
x = 0x0000006e, y = 0x0000033c
x = 0x00000067, y = 0x00000318
x = 0x0000002a, y = 0x0000039c
x = 0x0000004f, y = 0x00000354
x = 0x0000004d, y = 0x00000352
x = 0x00000042, y = 0x0000035a
x = 0x0000003d, y = 0x00000369
```


我们需要解决输出值不线性的问题

到底是触摸屏质量问题，还是 Adc 转化精度问题， 觉得应该是触摸屏电压不稳定 之前不知道 DELAY 寄存器是用来干嘛的

ADC START DELAY (ADCDLY) REGISTER

Register	Address	R/W	Description	Reset Value
ADCDLY	0x58000008	R/W	ADC start or interval delay register	0x00ff

ADCDLY	Bit	Description	Initial State
DELAY	[15:0]	1) Normal Conversion Mode, Separate X/Y Position Conversion Mode, and Auto (Sequential) X/Y Position Conversion Mode. → X/Y Position Conversion Delay Value. 2) Waiting for Interrupt Mode. When Stylus down occurs in Waiting for Interrupt Mode, this register generates Interrupt signal (INT_TC) at intervals of several ms for Auto X/Y Position conversion. **NOTE:** Do not use Zero value (0x0000)	00ff

等待中断模式时，当触摸笔按下时我们会产生中断，但是可以通过 DELAY 来延时产生中断

在前面有一张图

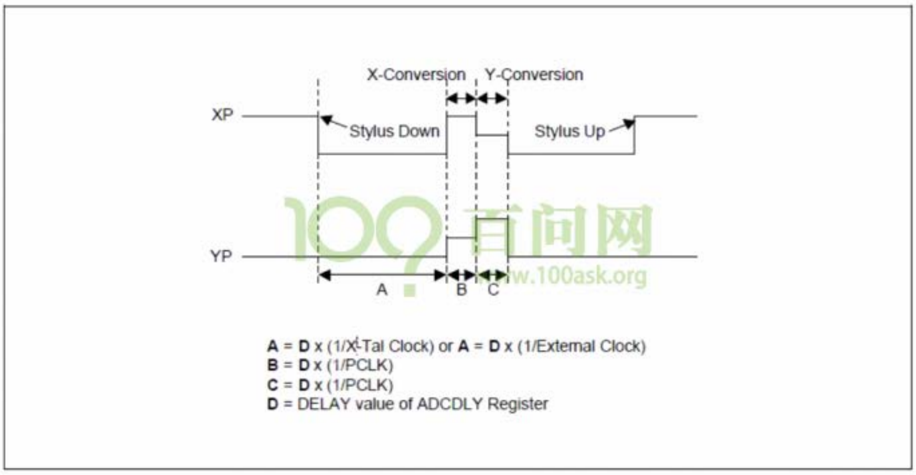


Figure 16-3 Timing Diagram in Auto (Sequential) X/Y Position Conversion Mode

按下触摸笔，延迟 A 才可以产生中断，你才可以测量 X Y 坐标 A = D（晶振的周期） D 就是 DELAY 就是那个寄存器的值 晶振周期时 12M 我们需要设置一下

```
void adc_ts_reg_init(void)
{
    /* [15] : ECFLG,  1 = End of A/D conversion
     * [14] : PRSCEN, 1 = A/D converter prescaler enable
     * [13:6]: PRSCVL, adc clk = PCLK / (PRSCVL + 1)
     * [5:3] : SEL_MUX, 000 = AIN 0
     * [2]   : STDBM
     * [0]   : 1 = A/D conversion starts and this bit is cleared after the startup.
     */
    ADCCON = (1<<14) | (49<<6) | (0<<3);

    /*  按下触摸屏, 延时一会再发出 TC 中断
```

```
        * 10ms 为120000
        *  延时时间 = ADCDLY * 晶振周期 = ADCDLY * 1 / 
12000000 = 5ms
        */
       ADCDLY = 60000;
    }
```

再次烧写，发现数据并不规律 我们需要再次改进程序

我们按下触摸屏会产生触摸屏中断，启动自动测量，启动 Adc，Adc 成功后会进入 Adc 中断，在函数中打印数据

也许测量过程很长 我们就需要判断

```
    void Isr_Adc(void)
    {
        int x = ADCDAT0;
        int y = ADCDAT1;
    //松开的话打印也是错误的值，所以如果仍然按下才打印
        if (!(x & (1<<15))) /* 如果仍然按下才打印 */
        {
                x &= 0x3ff;
                y &= 0x3ff;
                //打印10 进制
                printf("x = %08d, y = %08d\n\r", x, y);
        }

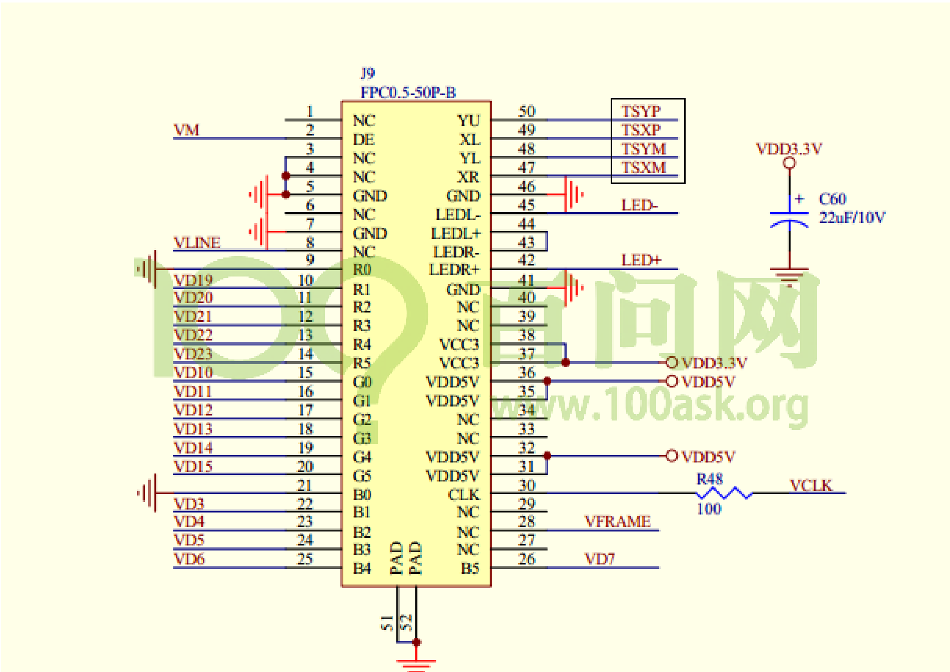
        enter_wait_pen_up_mode();
    }
```

烧写执行，发现 X Y 轴输出有问题

```
x = 00000192, y = 00000123
x = 00000193, y = 00000132
x = 00000184, y = 00000160
x = 00000197, y = 00000170
x = 00000205, y = 00000185
x = 00000181, y = 00000210
x = 00000182, y = 00000234
x = 00000201, y = 00000265
x = 00000202, y = 00000303
x = 00000194, y = 00000309
x = 00000202, y = 00000362
x = 00000195, y = 00000376
x = 00000207, y = 00000414
x = 00000220, y = 00000512
x = 00000212, y = 00000547
x = 00000191, y = 00000603
x = 00000192, y = 00000631
x = 00000196, y = 00000665
x = 00000223, y = 00000703
x = 00000222, y = 00000763
x = 00000227, y = 00000814
x = 00000247, y = 00000846
x = 00000246, y = 00000880
x = 00000236, y = 00000913
x = 00000263, y = 00000953
```


厂家把 X　Y 轴搞反了
电路图中

TSYP　TSXP 接反
TSYM　TSXM 接反

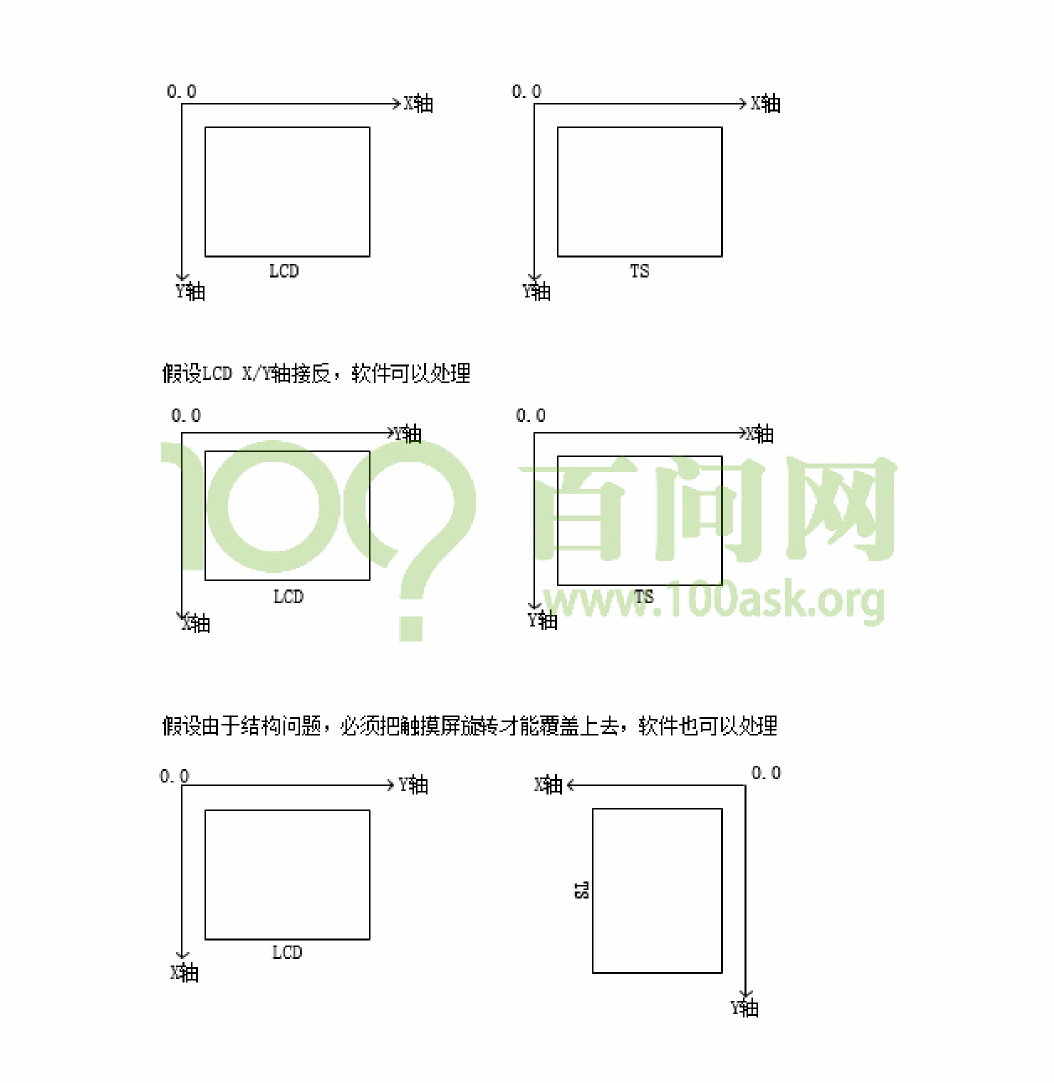


LCD接口插座

我们后面使用触摸屏时会使用软件处理这点，不会导致任何问题

有一个缺点 我们按下触摸屏会输出一个数据，再按下触摸屏又输出一个数据 我长按并没有输出数据，我滑动也没有输出数据 我们需要使用定时器改进这个问题

各种方向的旋转都可以由软件转换



我们需要把触摸屏的坐标 TS XY 坐标转换成 LCD 的 XY 坐标 需要用应用程序做 我们常使用 Tslib 库来做，这些旋转倒置都没有问题

第 007 节_触摸屏编程_定时器程序优化

- 有一个缺点
- 我们按下触摸屏会输出一个数据，再按下触摸屏又输出一个数据
- 我长按并没有输出数据，我滑动也没有输出数据
- 我们需要使用定时器改进这个问题

这个处理流程是怎么样的？

- 按下期间启动定时器
- 定时器每过 10ms / 20ms 就中断一次
- 在中断函数里测量触电的 XY 坐标
- 这样就可以得到连续的数据

打开定时器 Timer.c

```
#include "s3c2440_soc.h"

//定义一个宏 TIMER_NUM = 32

#define TIMER_NUM  32
#define NULL  ((void *)0)

typedef void(*timer_func)(void);

//定义一个结构体 ，既存放有函数指针又存放有数据

typedef struct timer_desc {
   char *name;
   timer_func fp;
}timer_desc, *p_timer_desc;

//我们需要往这个结构体数组里面添加函数，注册 Timer 函数

timer_desc timer_array[TIMER_NUM];



//注册 Timer 函数

int register_timer(char *name, timer_func fp)
{
   int i;
//搜索这个数组，如果 fp 等于 0 的话，就表示没有占用这个数组项，我就把它填充进去

   for (i = 0; i < TIMER_NUM; i++)
   {
         if (!timer_array[i].fp)
         {
               [i].name = name;
               timer_array[i].fp   = fp;
               return 0;
//注册成功
         }
   }
//否则，表示已经满了，注册失败
```

```
    return -1;
}
```

/*我们不需要使用 Timer 定时器的时候 unregister_timer 函数 考虑到我们需要从数组里面把这个 Timer 去掉，我们怎么找到这个 Timer？ 传入一个函数指针，以后卸载使用名字找到对应的项

- /

```
void unregister_timer(char *name)
{
//对于 unregister_timer 就反过来操作，遍历每一项
    int i;
    for (i = 0; i < TIMER_NUM; i++)
    {
//如果这个数组项里面的名字等于我传进来我名字
        if (!strcmp(timer_array[i].name, name))
        {
//也就表示我找到了这两项，设置成 NULL
            timer_array[i].name = NULL;
            timer_array[i].fp   = NULL;
            return 0;
        }
    }
//否则 return -1;找不到选择的项
    return -1;
}
```

应该让其从某个数组里面把需要定时器处理的函数依次执行，这样做，我们以后添加定时器处理函数时就不需要修改 Timer.c

```
void timer_irq(void)
{
    int i;
    for (i = 0; i < TIMER_NUM; i++)
    {
//判断指针是否为空，如果不是空的话就继续执行
timer_array[i].fp();这个函数
        if (timer_array[i].fp)
```

```
            {
                timer_array[i].fp();
            }
        }
    }
```

如果想继续点灯的话，需要单独注册 led_timer_irq 在 led.c 文件里注册 led_timer_irq

把这个函数放在 led_timer_irq 函数下面，防止编译错误 /*每 10ms 改函数被调用一次*/

```
int led_init(void)
{
    /* 设置GPFCON让GPF4/5/6配置为输出引脚 */
    GPFCON &= ~((3<<8) | (3<<10) | (3<<12));
    GPFCON |=  ((1<<8) | (1<<10) | (1<<12));
//led是名字，led_timer_irq是函数指针
    register_timer("led", led_timer_irq);
}



/* 每10ms 该函数被调用一次
 * 每500ms 操作一下LED实现计数
 */
void led_timer_irq(void)
{
    /* 点灯计数 */
    static int timer_num = 0;
    static int cnt = 0;
    int tmp;

    timer_num++;
    if (timer_num < 50)
        return;
    timer_num = 0;
//操作led
    cnt++;
```

```
        tmp = ~cnt;
        tmp &= 7;
        GPFDAT &= ~(7<<4);
        GPFDAT |= (tmp<<4);
    }
```

修改 main.c

```
    int main(void)
    {
        led_init();//初始化 led

        //interrupt_init();  /* 初始化中断控制器 */

        key_eint_init();   /* 初始化按键, 设为中断源 */

        timer_init();//打开定时器


        puts("\n\rg_A = ");
        printHex(g_A);
        puts("\n\r");

        //nor_flash_test();
        lcd_test();

        //adc_test();

        touchscreen_test();

        while (1);

        return 0;
    }
```

修改 timer.c 文件

```
    void timer_init(void)
    {
        /* 设置 TIMER0 的时钟 修改时钟频录让其 10ms 中断一次 */

        /* Timer clk = PCLK / {prescaler value+1} /
{divider value}
                    = 50000000/(49+1)/16
```

```
               = 62500
     */

    TCFG0 = 49;  /* Prescaler 0 = 49, 用于timer0,1 */

    TCFG1 &= ~0xf;
    TCFG1 |= 3;  /* MUX0 : 1/16 */

    /* 设置TIMER0的初值 */

    TCNTB0 = 625;  /* 10Ms中断一次 */

    /* 加载初值, 启动timer0 */

    TCON |= (1<<1);   /* Update from TCNTB0 & TCMPB0 */

    /* 设置为自动加载并启动 */

    TCON &= ~(1<<1);
    TCON |= (1<<0) | (1<<3);  /* bit0: start, bit3: auto
reload */

    /* 设置中断 */

    register_irq(10, timer_irq);
}
```

烧写到nandflash发现无输出，可能是前重定位前的代码超出了4k，所以我们使用Norflash启动 发现可以正常运行

我们修改Makefile把负责重定位代码往前移，其他无关代码往后放

看star.S

```
.text
.global _start

_start:
    b reset          /* vector 0 : reset */
    ldr pc, und_addr /* vector 4 : und */
    ldr pc, swi_addr /* vector 8 : swi */
    b halt               /* vector 0x0c : prefetch aboot
*/
    b halt               /* vector 0x10 : data abort */
    b halt               /* vector 0x14 : reserved */
    ldr pc, irq_addr /* vector 0x18 : irq */
    b halt               /* vector 0x1c : fiq */
```

```
//我们需要把这些异常往后放
und_addr:
    .word do_und

swi_addr:
    .word do_swi

irq_addr:
    .word do_irq



reset:
    /* 关闭看门狗 */
    ldr r0, =0x53000000
    ldr r1, =0
    str r1, [r0]

    /* 设置MPLL, FCLK : HCLK : PCLK = 400m : 100m : 50m
*/
    /* LOCKTIME(0x4C000000) = 0xFFFFFFFF */
    ldr r0, =0x4C000000
    ldr r1, =0xFFFFFFFF
    str r1, [r0]

    /* CLKDIVN(0x4C000014) = 0X5, tFCLK:tHCLK:tPCLK =
1:4:8  */
    ldr r0, =0x4C000014
    ldr r1, =0x5
    str r1, [r0]

    /* 设置CPU工作于异步模式 */
    mrc p15,0,r0,c1,c0,0
    orr r0,r0,#0xc0000000   //R1_nF:OR:R1_iA
    mcr p15,0,r0,c1,c0,0

    /* 设置MPLLCON(0x4C000004) = (92<<12)|(1<<4)|(1<<0)
     *  m = MDIV+8 = 92+8=100
```

```
 *  p = PDIV+2 = 1+2 = 3
 *  s = SDIV = 1
 *  FCLK = 2*m*Fin/(p*2^s) = 2*100*12/(3*2^1)=400M
 */
ldr r0, =0x4C000004
ldr r1, =(92<<12)|(1<<4)|(1<<0)
str r1, [r0]

/* 一旦设置PLL, 就会锁定lock time直到PLL输出稳定
 * 然后CPU工作于新的频率FCLK
 */



/* 设置内存: sp 栈 */
/* 分辨是nor/nand启动
 * 写0到0地址, 再读出来
 * 如果得到0, 表示0地址上的内容被修改了, 它对应ram, 这就
是nand启动
 * 否则就是nor启动
 */
mov r1, #0
ldr r0, [r1] /* 读出原来的值备份 */
str r1, [r1] /* 0->[0] */
ldr r2, [r1] /* r2=[0] */
cmp r1, r2   /* r1==r2? 如果相等表示是NAND启动 */
ldr sp, =0x40000000+4096 /* 先假设是nor启动 */
moveq sp, #4096  /* nand启动 */
streq r0, [r1]   /* 恢复原来的值 */

bl sdram_init
```

```
    //bl sdram_init2    /* 用到有初始值的数组, 不是位置无关码
*/

    /* 重定位 text, rodata, data 段整个程序 */
    bl copy2sdram

    /* 清除 BSS 段 */
    bl clean_bss

    /* 复位之后, cpu 处于 svc 模式
     * 现在, 切换到 usr 模式
     */
    mrs r0, cpsr        /* 读出 cpsr */

    bic r0, r0, #0xf     /* 修改 M4-M0 为 0b10000, 进入 usr
模式 */

    bic r0, r0, #(1<<7)  /* 清除 I 位, 使能中断 */
    msr cpsr, r0

    /* 设置 sp_usr */
    ldr sp, =0x33f00000

    ldr pc, =sdram
sdram:
    bl uart0_init

    bl print1
    /* 故意加入一条未定义指令 */
```

init.c 需要放在前面
nand_init
sdram_init 初始化也放前面

sdram: bl uart0_init 这就跳到 sdram 了，重定位后就随便操作

修改 Makefile 我们把 start.o init.o nand_flash.o 放在最前面

```
objs = start.o init.o nand_flash.o led.o uart.o main.o exception.o interrupt.o timer.o nor_flash.o my_printf.o string_utils.o lib1funcs.o

objs += lcd/font.o
objs += lcd/framebuffer.o
objs += lcd/geometry.o
objs += lcd/lcd.o
objs += lcd/lcd_4.3.o
objs += lcd/lcd_controller.o
objs += lcd/lcd_test.o
objs += lcd/s3c2440_lcd_controller.o
objs += lcd/font_8x16.o

objs += adc_touchscreen/adc.o
objs += adc_touchscreen/adc_test.o

objs += adc_touchscreen/touchscreen.o
objs += adc_touchscreen/touchscreen_test.o


all: $(objs)
	#arm-linux-ld -Ttext 0 -Tdata 0x30000000  start.o led.o uart.o init.o main.o -o sdram.elf
	arm-linux-ld -T sdram.lds $^ libgcc.a -o sdram.elf
	arm-linux-objcopy -O binary -S sdram.elf sdram.bin
	arm-linux-objdump -D sdram.elf > sdram.dis
clean:
	rm -f *.bin $(objs) *.elf *.dis

%.o : %.c
	arm-linux-gcc -march=armv4 -c -o $@ $<

%.o : %.S
	arm-linux-gcc -march=armv4 -c -o $@ $<
```

这节课讲定时器的优化 下节课讲怎么使用定时器来改进触摸屏

第 008 节_触摸屏编程_使用定时器支持长按

可以使用定时器把长按或者滑动触摸屏的值读出来

我们按下触摸屏就会产生触摸屏中断，这个时候可以启动 ADC ADC 成功后再次产生中段 在这个中断中启动定时器

我们也可以在触摸屏中断里启动定时器

```
#include "../s3c2440_soc.h"

#define ADC_INT_BIT (10)
#define TC_INT_BIT  (9)

#define INT_ADC_TC   (31)


/* ADCTSC's bits */
#define WAIT_PEN_DOWN    (0<<8)
#define WAIT_PEN_UP      (1<<8)

#define YM_ENABLE        (1<<7)
#define YM_DISABLE       (0<<7)

#define YP_ENABLE        (0<<6)
#define YP_DISABLE       (1<<6)

#define XM_ENABLE        (1<<5)
#define XM_DISABLE       (0<<5)

#define XP_ENABLE        (0<<4)
#define XP_DISABLE       (1<<4)

#define PULLUP_ENABLE    (0<<3)
#define PULLUP_DISABLE   (1<<3)

#define AUTO_PST         (1<<2)

#define WAIT_INT_MODE    (3)
#define NO_OPR_MODE      (0)


//定义一个全局变量设置 timer 状态
static volatile int g_ts_timer_enable = 0;


void adc_ts_int_init(void)
{
```

```
        SUBSRCPND = (1<<TC_INT_BIT) | (1<<ADC_INT_BIT);

     //注册中断处理函数，才可以在处理函数中启动定时器

        /* 注册中断处理函数 */
        register_irq(31, AdcTsIntHandle);

        /* 使能中断 */
        INTSUBMSK &= ~((1<<ADC_INT_BIT) | (1<<TC_INT_BIT));
        //INTMSK    &= ~(1<<INT_ADC_TC);
    }

    void touchscreen_init(void)
    {
        /* 设置触摸屏接口:寄存器 */
        adc_ts_reg_init();

        printf("ADCUPDN = 0x%x, SUBSRCPND = 0x%x, SRCPND =
0x%x\n\r", ADCUPDN, SUBSRCPND, SRCPND);

        /* 设置中断 */
        adc_ts_int_init();

        /* 注册定时器处理函数 */
    //首先是一个名字，其次是定时器处理函数
        register_timer("touchscreen",
touchscreen_timer_irq);

        /* 让触摸屏控制器进入"等待中断模式" */
        enter_wait_pen_down_mode();
    }
```

进入 touchscreen.c adc 中断处理函数

```
    void Isr_Adc(void)
    {
        int x = ADCDAT0;
```

```c
    int y = ADCDAT1;

    if (!(x & (1<<15))) /* 如果仍然按下才打印 */
    {
            x &= 0x3ff;
            y &= 0x3ff;

            printf("x = %08d, y = %08d\n\r", x, y);

//添加定时器函数

            /* 启动定时器以再次读取数据 */
            ts_timer_enable();
    }
//松开操作
    else
    {
            ts_timer_disable();
            enter_wait_pen_down_mode();
    }

    enter_wait_pen_up_mode();
}

//触摸屏定时器处理函数

/* 每10ms 该函数被调用一次
 */
void touchscreen_timer_irq(void)
{
    /* 如果触摸屏仍被按下, 进入"自动测量模式", 启动ADC */

//如果定时器并没有被使能, 则return
    if (get_status_of_ts_timer() == 0)
            return;

//如果松开, 则什么事情都不做

    if (ADCDAT0 & (1<<15)) /* 如果松开 */
```

```
    {
//设置定时器状态
            ts_timer_disable();
//触摸笔进入等待模式
            enter_wait_pen_down_mode();
            return;
    }

//否则启动测量

    else  /* 按下状态，启动下一次测量 */
    {
            /* 进入"自动测量"模式 */
            enter_auto_measure_mode();

            /* 启动ADC  */
            ADCCON |= (1<<0);
    }
}


static void ts_timer_enable(void)
{

//我们使用定时器时把它的状态设置为1
    g_ts_timer_enable = 1;
}



//有启用定时器就有关闭定时器
static void ts_timer_disable(void)
{

//我们不使用定时器把定时器状态设置为0
    g_ts_timer_enable = 0;
}
```

```c
//我们如何获取定时器状态？
static int get_status_of_ts_timer(void)
{
//返回定时器状态
    return g_ts_timer_enable;
}
```

我们回顾一下处理过程，根据流程图分析
首先从 main.c 函数开始

```c
int main(void)
{
    led_init();
    //interrupt_init();  /* 初始化中断控制器 */

    key_eint_init();   /* 初始化按键，设为中断源 */
    timer_init();

    puts("\n\rg_A = ");
    printHex(g_A);
    puts("\n\r");

    //nor_flash_test();
    lcd_test();

    //adc_test();
//执行 touchscreen_test
    touchscreen_test();

    while (1);

    return 0;
}
```

进入 touchscreen_test.c 文件执行 init 初始化程序

```c
void touchscreen_test(void)
{
    touchscreen_init();
```

```
}
```

进入 touchscreen.c 文件

```
void touchscreen_init(void)
{
    /* 设置触摸屏接口:寄存器 */
    adc_ts_reg_init();

    printf("ADCUPDN = 0x%x, SUBSRCPND = 0x%x, SRCPND = 0x%x\n\r", ADCUPDN, SUBSRCPND, SRCPND);

    /* 设置中断 */
    adc_ts_int_init();

    /* 让触摸屏控制器进入"等待中断模式" */
    enter_wait_pen_down_mode();
}


void adc_ts_reg_init(void)
{
    /* [15] : ECFLG,  1 = End of A/D conversion
     * [14] : PRSCEN, 1 = A/D converter prescaler enable
     * [13:6]: PRSCVL, adc clk = PCLK / (PRSCVL + 1)
     * [5:3] : SEL_MUX, 000 = AIN 0
     * [2]   : STDBM
     * [0]   : 1 = A/D conversion starts and this bit is cleared after the startup.
     */
    ADCCON = (1<<14) | (49<<6) | (0<<3);

    /*  按下触摸屏, 延时一会再发出 TC 中断

     *  延时时间 = ADCDLY * 晶振周期 = ADCDLY * 1 / 12000000 = 5ms
     */
    ADCDLY = 60000;
}
```

```c
//进入 adc 触摸屏中断处理
void adc_ts_int_init(void)
{
    SUBSRCPND = (1<<TC_INT_BIT) | (1<<ADC_INT_BIT);

    /* 注册中断处理函数 */
    register_irq(31, AdcTsIntHandle);

    /* 使能中断 */
    INTSUBMSK &= ~((1<<ADC_INT_BIT) | (1<<TC_INT_BIT));
    //INTMSK    &= ~(1<<INT_ADC_TC);
}


void AdcTsIntHandle(int irq)
{
    if (SUBSRCPND & (1<<TC_INT_BIT))  /* 如果是触摸屏中断
*/
            Isr_Tc();

    if (SUBSRCPND & (1<<ADC_INT_BIT))  /* ADC 中断 */
            Isr_Adc();
    SUBSRCPND = (1<<TC_INT_BIT) | (1<<ADC_INT_BIT);
}


//进入触摸屏处理函数

//触摸屏定时器处理函数

/* 每 10ms 该函数被调用一次
 */
void touchscreen_timer_irq(void)
{
    /* 如果触摸屏仍被按下, 进入"自动测量模式", 启动 ADC */

//如果定时器并没有被使能, 则 return
```

```
        if (get_status_of_ts_timer() == 0)
                return;
    //如果松开，则什么事情都不做

        if (ADCDAT0 & (1<<15)) /* 如果松开 */
        {
    //设置定时器状态
                ts_timer_disable();
    //触摸笔进入等待模式
                enter_wait_pen_down_mode();
                return;
        }

    //否则启动测量

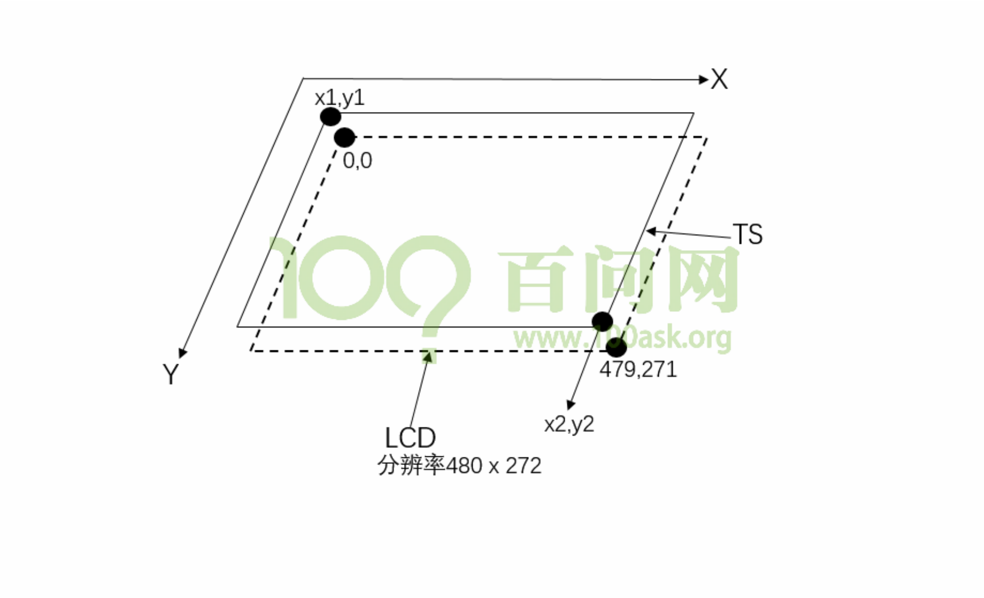
        else  /* 按下状态，启动下一次测量 */
        {
                /* 进入"自动测量"模式 */
                enter_auto_measure_mode();

                /* 启动ADC  */
                ADCCON |= (1<<0);
        }
    }
```

第 009 节_触摸屏编程_较准原理

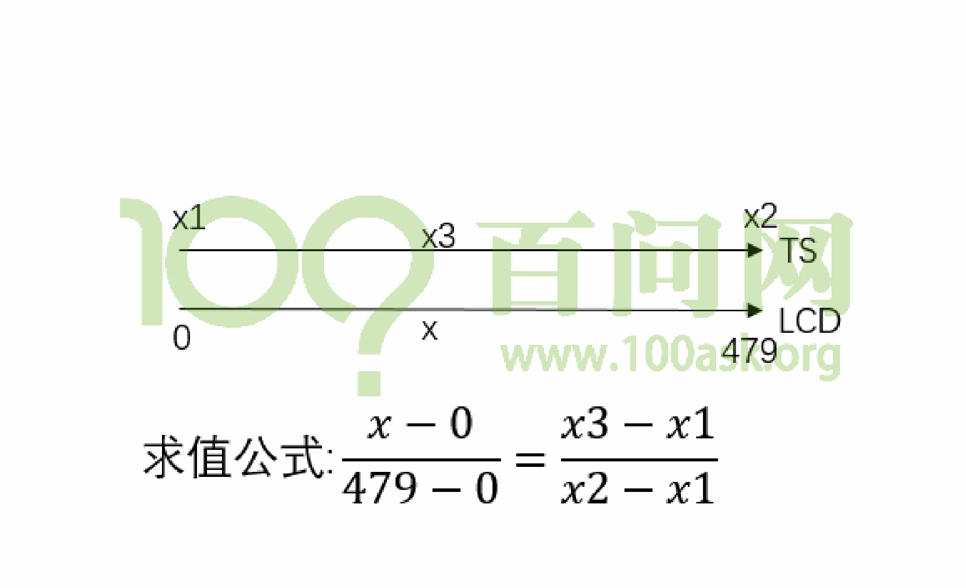
我们需要校准触摸屏，所谓校准就是找到一个公式把电压值转换为坐标值

触摸屏和 LCD 是两个东西 触摸屏覆盖在 LCD



上

问：得到触电的(x1,y1)怎么换算出 LCD 的坐标值 (X,Y)



x=479-0/x2-x1 * (x3-x1) + 0

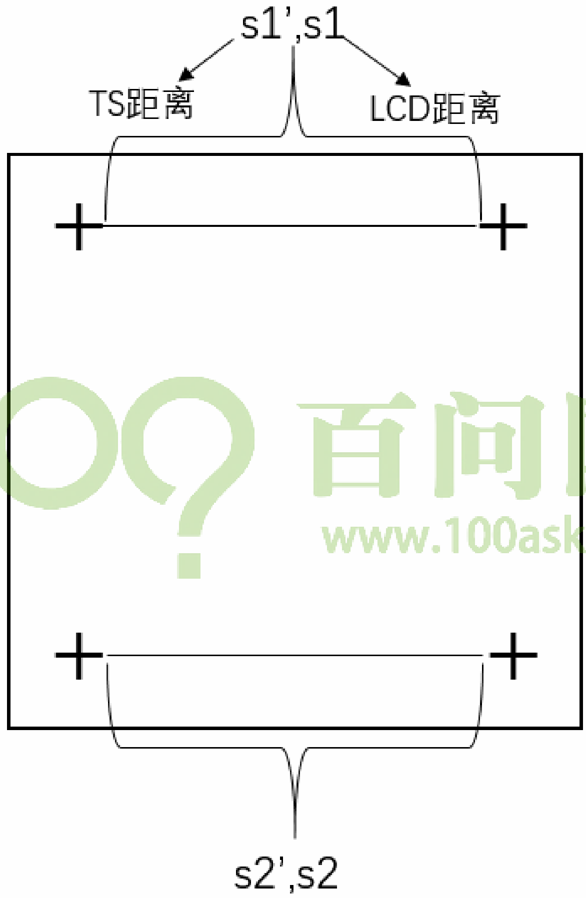
x=长度的比例

x1 0 原点的触摸屏/LCD 坐标

我们只需要确定两个点就可以把 lcd 坐标确定下来，但是我们可以做的更好

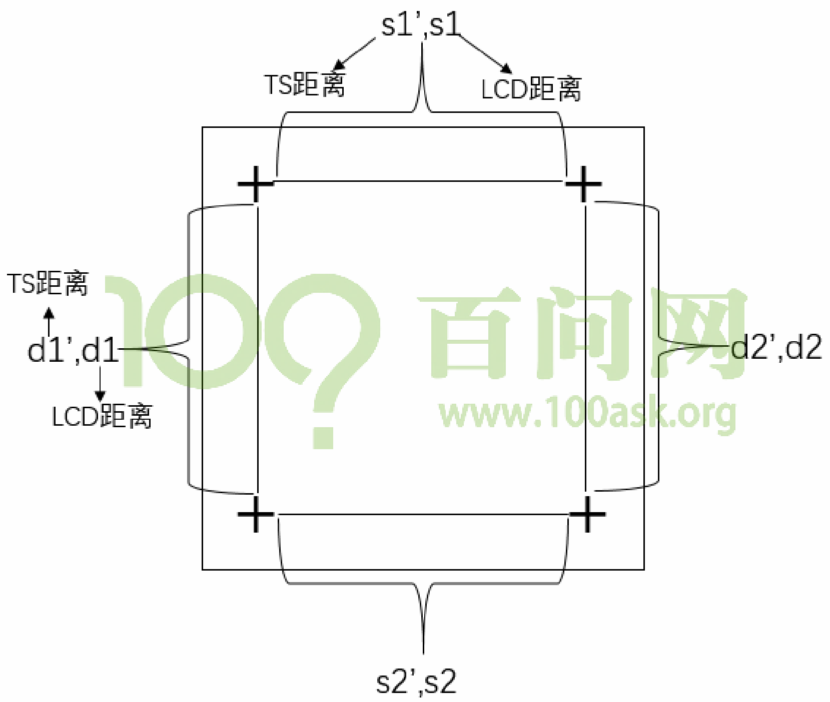
假设由于制作工艺问题，导致触摸屏和 LCD 坐标并不相同，需要其他公式计算

X 轴方向



s1'是 TS 上 X 轴两个点的距离
s1 是 LCD 上 X 轴两个点的距离
s2' s2
Kx= LCD 距离/触摸屏距离
= (s1 + s2) / (s1' + s2')
= 2s/(s1' + s2')

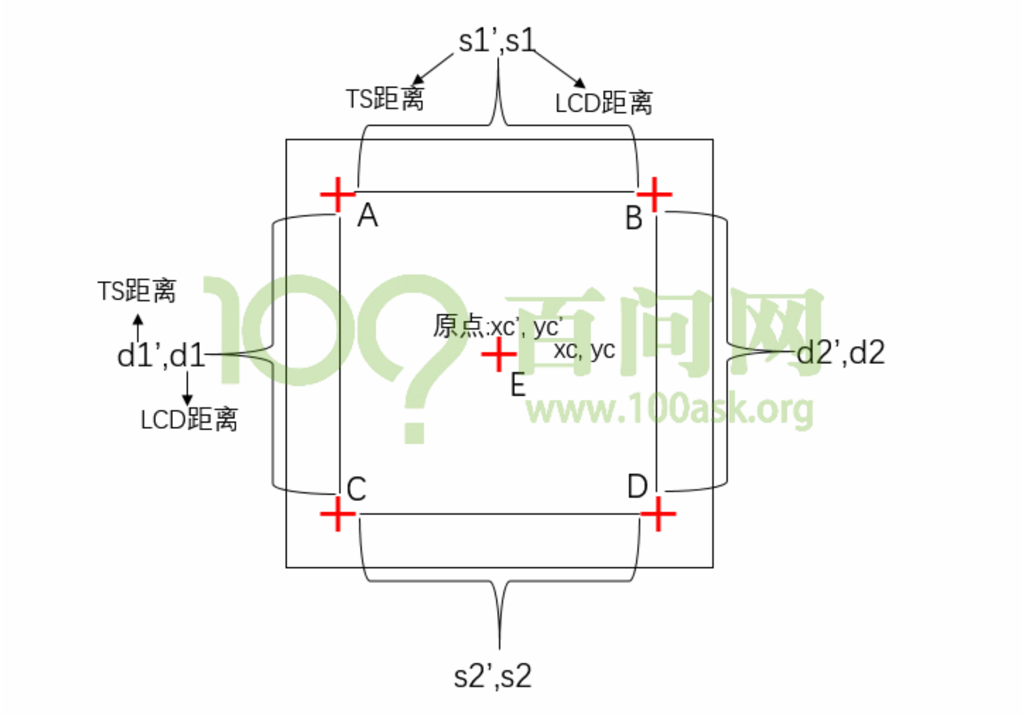
Y 轴方向，



TS 距离是 d1'
LCD 距离是 d1
Ky=(d1 + d2) / (d1' + d2')
= 2d / (d1' + d2')

我们现在有了斜率，给定一个坐标，我们需要需要原点的触屏 LCD 坐标

原点我们选在最中间 可以忽略掉上下左右的偏差



原点坐标在触摸屏上是 xc' yc' 在 LCD 上是 xc yc 那我们的校准公式 对于给定的 x3,我们如何求出 x

X= (x3 - xc') * Kx + xc

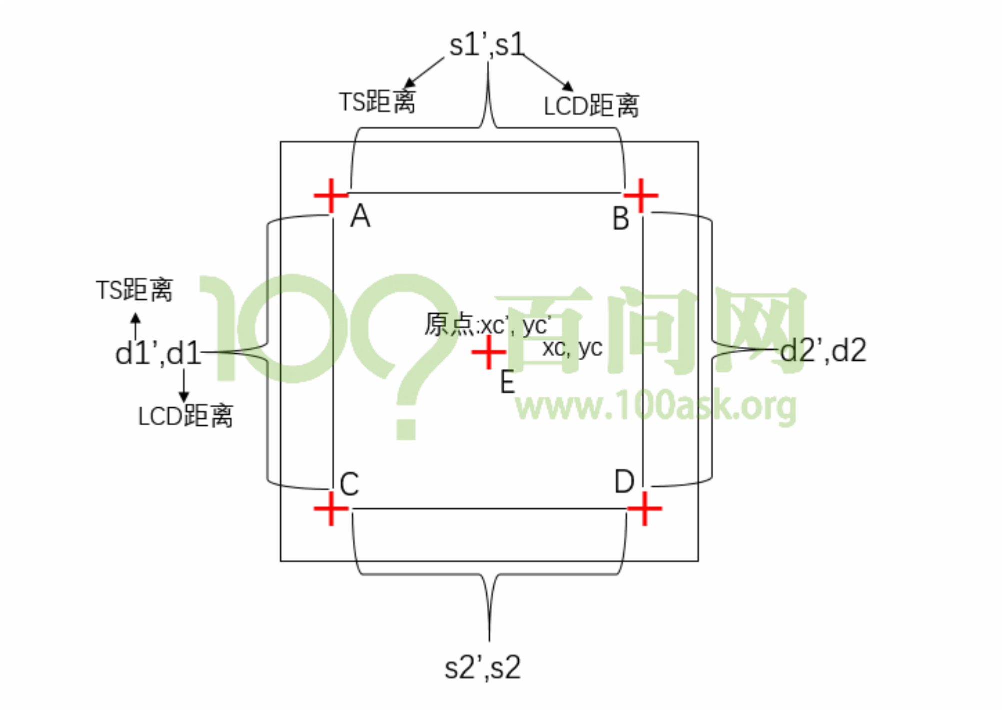
对于给定的 y'我们如何算出 Y 轴坐标?

y = (y' - yc') * Ky + yc

我们需要点击触摸屏上这 5 个点，同时需要把这五个点坐标打印显示出来这节视频我们讲的时校准原理

第 010 节_触摸屏编程_较准与画线编

这个程序我们怎么写



我们需要得到这 5 个点的坐标 给这 5 个点分别设置为 ABCDE

第一步

1. 在 A 点显示 “ + ”
2. 客户点击 “ + ”
3. 记录触摸屏的坐标
4. 在 BCDE 上循环操作,显示点击读取的操作

第二步 根据这些数据，确定公式

第三步 以后得到 TS 触点时，可转换出 LCD 坐标

我们需要实现这几个函数

显示” + ” 在 x y 中显示 fb_disp_cross(int x , int y)

如何记录 ts_read_raw 读到原始数据

根据这些数据，确定公式

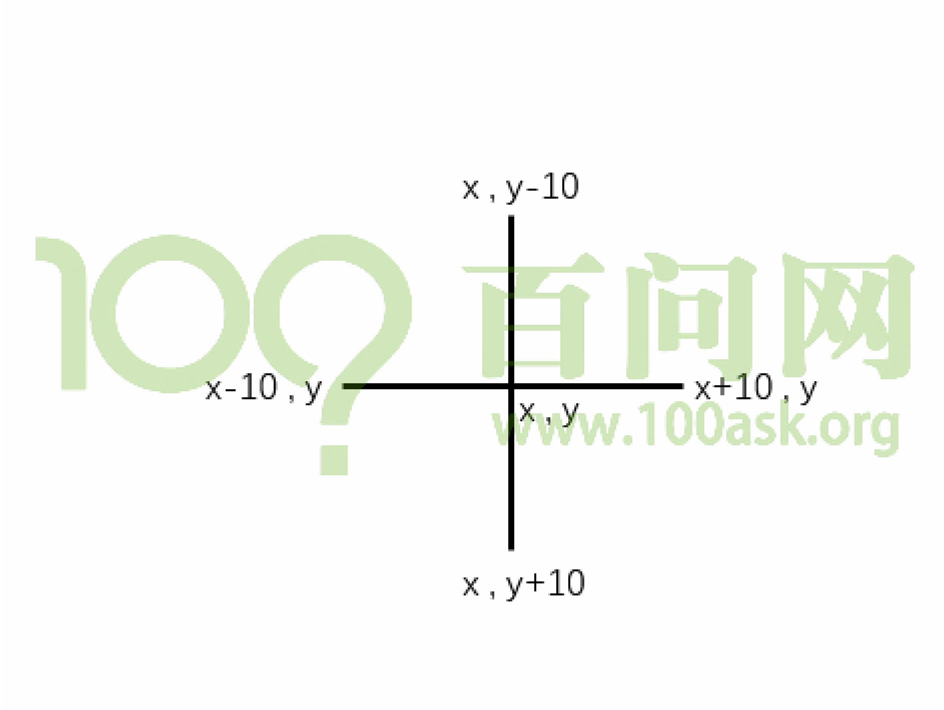
ts_calibrate

如何转换出 LCD 坐标 ts_read

我们实现这几个函数

我们先实现 “ + ”

我们既然画线就在 geometry.c 中实现



画十字架 原点 x,y x-10 y x+10 y x y+10 x y+10

void fb_disp_cross(int x, int y, unsigned int color)
{ draw_line(x-10, y, x+10, y, color); draw_line(x, y-10, x, y+10, color); }

编辑 touchscreen.c

```
//+ 坐标的 x 值
static int g_ts_x;
//+ 坐标 Y 值
static int g_ts_y;

//表示数据并未有效

//设置成 volatile 类型，有两个地方会用到一个是中断
report_ts_xy
//另一个是程序 ts_read_raw，我们一定确保这个值是从内存中读取出
来
//让双方得到真实的值
static  volatile char g_ts_data_valid = 0;
```

```
void Isr_Adc(void)
{
    int x = ADCDAT0;
    int y = ADCDAT1;

    if (!(x & (1<<15))) /* 如果仍然按下才打印 */
    {
            x &= 0x3ff;
            y &= 0x3ff;
//我们现在不能打印
            //printf("x = %08d, y = %08d\n\r", x, y);
//实现 report_ts_xy 函数来打印
            report_ts_xy(x, y);

            /* 启动定时器以再次读取数据 */
            ts_timer_enable();
    }
    else
    {
            ts_timer_disable();
            enter_wait_pen_down_mode();
    }

    enter_wait_pen_up_mode();
}

//report_ts_xy 函数的实现

void report_ts_xy(int x, int y)
{
    //printf("x = %08d, y = %08d\n\r", x, y);
//一开始标记位=0 表示没有数据
    if (g_ts_data_valid == 0)
    {
            g_ts_x = x;
```

```
            g_ts_y = y;
            g_ts_data_valid = 1;
    }
}

//读到原始数据
void ts_read_raw(int *px, int *py)
{
//当按下触摸屏时会产生 ADC 中断
    while (g_ts_data_valid == 0);
    *px = g_ts_x;
    *py = g_ts_y;
//读完数据清零
    g_ts_data_valid = 0;
}
```

下面我们实现 ts_calibrate 校准函数
tslib.c 文件中

```
//设置斜率为全局变量
static double g_kx;
static double g_ky;

static int g_ts_xc, g_ts_yc;
static int g_lcd_xc, g_lcd_yc;


//加一个调换的全局变量
static int g_ts_xy_swap = 0;


/*
----------------------------
|                       |
|  +(A)            (B)+  |
|                       |
|                       |
|                       |
|          +(E)          |
```

```
|                      |
|                      |
|                      |
|  +(D)          (C)+  |
|                      |
----------------------------

*/
```

把 ABCDE 这 5 个点都实现

```
void ts_calibrate(void)
{
   unsigned int fb_base;
   int xres, yres, bpp;
//定义 ABCDE 触摸屏坐标
   int a_ts_x, a_ts_y;
   int b_ts_x, b_ts_y;
   int c_ts_x, c_ts_y;
   int d_ts_x, d_ts_y;
   int e_ts_x, e_ts_y;

   /* X 轴方向 */
   int ts_s1, ts_s2;
   int lcd_s;

   /* Y 轴方向 */
   int ts_d1, ts_d2;
   int lcd_d;
//通过调用 framebuffer.c 里面的函数，来获取 LCD 坐标
   /* 获得 LCD 的参数: fb_base, xres, yres, bpp */
   get_lcd_params(&fb_base, &xres, &yres, &bpp);

   /* 对于 ABCDE, 循环: 显示"+"、点击、读 ts 原始值 */
//A 坐标，X 分辨率 50，  Y 分辨率 50
   /* A(50, 50) */
   get_calibrate_point_data(50, 50, &a_ts_x, &a_ts_y);
```

```
    //B 坐标，X 分辨率-50
        /* B(xres-50, 50) */
        get_calibrate_point_data(xres-50, 50, &b_ts_x,
&b_ts_y);
    //C 坐标，X 分辨率-50，Y 分辨率-50
        /* C(xres-50, yres-50) */
        get_calibrate_point_data(xres-50, yres-50, &c_ts_x,
&c_ts_y);
    //D 坐标，X 分辨率位置 50，Y 分辨率-50
        /* D(50, yres-50) */
        get_calibrate_point_data(50, yres-50, &d_ts_x,
&d_ts_y);
    //E 坐标，X 分辨率除 2，Y 分辨率除 2
        /* E(xres/2, yres/2) */
        get_calibrate_point_data(xres/2, yres/2, &e_ts_x,
&e_ts_y);

    //读取 XY 坐标值后确定 XY 是否反转

        /* 确定触摸屏数据 XY 是否反转 */

        g_ts_xy_swap = is_ts_xy_swap(a_ts_x, a_ts_y,
b_ts_x, b_ts_y);
    //如果反转，对调所有点的 XY 坐标
        if (g_ts_xy_swap)
        {
              /* 对调所有点的 XY 坐标 */
              swap_xy(&a_ts_x, &a_ts_y);
              swap_xy(&b_ts_x, &b_ts_y);
              swap_xy(&c_ts_x, &c_ts_y);
              swap_xy(&d_ts_x, &d_ts_y);
              swap_xy(&e_ts_x, &e_ts_y);
        }
```




```
    /* 确定公式的参数并保存 */
//Ts_S1 值=B 点 ts 和 A 点 tsX 轴方向的距离
    ts_s1 = b_ts_x - a_ts_x;
//Ts_S2 值
    ts_s2 = c_ts_x - d_ts_x;
//lcd_s 值
    lcd_s = xres-50 - 50;
//ts_d1 值
    ts_d1 = d_ts_y - a_ts_y;
//ts_d2 值
    ts_d2 = c_ts_y - b_ts_y;
//lcd_d 值
    lcd_d = yres-50-50;
//X 轴的斜率
```

```
    g_kx = ((double)(2*lcd_s)) / (ts_s1 + ts_s2);
//Y 轴的斜率
    g_ky = ((double)(2*lcd_d)) / (ts_d1 + ts_d2);

//中心点 E 点的坐标
    g_ts_xc = e_ts_x;

    g_ts_yc = e_ts_y;

    g_lcd_xc = xres/2;
    g_lcd_yc = yres/2;
}
```

我们需要把 + 在 LCD 上显示出来并且读出数据

```
void get_calibrate_point_data(int lcd_x, int lcd_y, int *px, int *py)
{
    fb_disp_cross(lcd_x, lcd_y, 0xffffff);


    /* 等待点击 */


    ts_read_raw(px, py);
}
```

比如我们之前发现上报的 X 轴 Y 轴值反了

比如正常情况下从 A 点移动到 B 点是 x 值变化比较大 y 值不变，但是目前的情况是 y 值变化比较大，x 值不变

我分根据这个特性分辨

```
int is_ts_xy_swap(int a_ts_x, int a_ts_y, int b_ts_x, int b_ts_y)
{
    int dx = b_ts_x - a_ts_x;
    int dy = b_ts_y - a_ts_y;
//减出来的值有可能是负数，我们需要取绝对值
    if (dx < 0)
            dx = 0 - dx;
    if (dy < 0)
```

```
		dy = 0 - dy;

	if(dx > dy)
		return 0; /* xy 没有反转 */
	else
		return 1; /* xy 反了 */
}
```

如果是反的我们需要调换回来，我们需要确定 XY 是否反转

```
//我们写出一个对调函数
void swap_xy(int *px, int *py)
{
	int tmp = *px;
	*px = *py;
	*py = tmp;
}

/*
 * 读 TS 原始数据, 转换为 LCD 坐标
 */
void ts_read(int *lcd_x, int *lcd_y)
{
	int ts_x, ts_y;

	ts_read_raw(&ts_x, ts_y);

	if (g_ts_xy_swap)
	{
		swap_xy(&ts_x, &ts_y);
	}

	/* 使用公式计算 */
//斜率 * (触摸屏坐标 - 中心点除，触摸屏坐标) + 中心点 LCD 坐标
	*lcd_x = g_kx * (ts_x - g_ts_xc) + g_lcd_xc;
	*lcd_y = g_ky * (ts_y - g_ts_yc) + g_lcd_yc;
}
```

//接下来我们画线 我们在 framebuffer.c 中把清屏函数单独实现

```c
void clear_screen(unsigned int color)
{
	int x, y;
	unsigned char *p0;
	unsigned short *p;
	unsigned int *p2;

	/* 往 framebuffer 中写数据 */
	if (bpp == 8)
	{
		/* bpp: palette[color] */

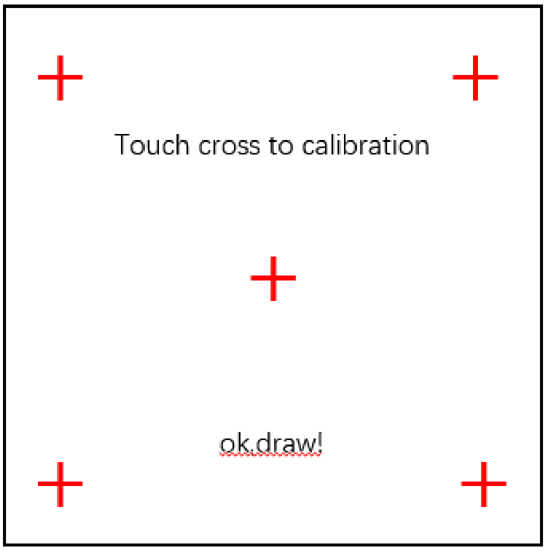
		p0 = (unsigned char *)fb_base;
		for (x = 0; x < xres; x++)
			for (y = 0; y < yres; y++)
				*p0++ = color;
	}
	else if (bpp == 16)
	{
		/* 让 LCD 输出整屏的红色 */

		/* 565: 0xf800 */

		p = (unsigned short *)fb_base;
		for (x = 0; x < xres; x++)
			for (y = 0; y < yres; y++)
				*p++ =
convert32bppto16bpp(color);

	}
	else if (bpp == 32)
	{
		p2 = (unsigned int *)fb_base;
		for (x = 0; x < xres; x++)
			for (y = 0; y < yres; y++)
				*p2++ = color;

	}
}
```



`![](                                                                                      )`
先显示一个一个点，让后显示开始校准提示， 校准完提示 ok draw
打开我们的 touchscreen_test.c 文件

```
void touchscreen_test(void)
{
    unsigned int fb_base;
    int xres, yres, bpp;

    int x, y;


    /* 获得LCD 的参数: fb_base, xres, yres, bpp */
    get_lcd_params(&fb_base, &xres, &yres, &bpp);

    touchscreen_init();


    /* 清屏 */
    clear_screen(0);
//我们在70 像素的地方显示文字，背景白色显示文字

    /* 显示文字提示较准 */
```

```
        fb_print_string(70, 70, "Touc cross to calibrate
touchscreen", 0xffffff);
        ts_calibrate();

        /* 显示文字提示绘画 */
        fb_print_string(70, yres - 70, "OK! To draw!",
0xffffff);

        while (1)
        {
                ts_read(&x, &y);
    //我们先打印值
                printf(" x = %d, y = %d\n\r", x, y);
    //描绿色的线
                fb_put_pixel(x, y, 0xff00);
        }
    }
```

我们添加了 tslib.c 需要修改 Makefile
添加 objs += adc_touchscreen/tslib.o

```
    objs = start.o init.o nand_flash.o led.o uart.o main.o
exception.o interrupt.o timer.o nor_flash.o my_printf.o
string_utils.o lib1funcs.o

    objs += lcd/font.o
    objs += lcd/framebuffer.o
    objs += lcd/geometry.o
    objs += lcd/lcd.o
    objs += lcd/lcd_4.3.o
    objs += lcd/lcd_controller.o
    objs += lcd/lcd_test.o
    objs += lcd/s3c2440_lcd_controller.o
    objs += lcd/font_8x16.o

    objs += adc_touchscreen/adc.o
    objs += adc_touchscreen/adc_test.o

    objs += adc_touchscreen/touchscreen.o
    objs += adc_touchscreen/touchscreen_test.o
```

```
objs += adc_touchscreen/tslib.o


all: $(objs)
	#arm-linux-ld -Ttext 0 -Tdata 0x30000000  start.o led.o uart.o init.o main.o -o sdram.elf
	arm-linux-ld -T sdram.lds $^ libgcc.a -o sdram.elf
	arm-linux-objcopy -O binary -S sdram.elf sdram.bin
	arm-linux-objdump -D sdram.elf > sdram.dis
clean:
	rm -f *.bin $(objs) *.elf *.dis

%.o : %.c
	arm-linux-gcc -march=armv4 -c -o $@ $<

%.o : %.S
	arm-linux-gcc -march=armv4 -c -o $@ $<
```

第 011 节_触摸屏编程_测试

发现程序有 bug,点击坐标一次，程序就完成执行 我们需要修改触摸屏文件 tsib.c

```
static double g_kx;
static double g_ky;

static int g_ts_xc, g_ts_yc;
static int g_lcd_xc, g_lcd_yc;
static int g_ts_xy_swap = 0;


static unsigned int fb_base;
static int xres, yres, bpp;


void get_calibrate_point_data(int lcd_x, int lcd_y, int *px, int *py)
{
	int pressure;
	int x, y;
```

```
        fb_disp_cross(lcd_x, lcd_y, 0xffffff);

        /* 等待点击 */

        do {
    //如果 pressure 一直是 0 的话我们丢掉这些数据
                ts_read_raw(&x, &y, &pressure);
        } while (pressure == 0);

    //让后再次读
        do {
                *px = x;
                *py = y;
                ts_read_raw(&x, &y, &pressure);
                printf("get raw data: x = %08d, y
= %08d\n\r", x, y);
        } while (pressure);

        printf("return raw data: x = %08d, y = %08d\n\r",
*px, *py);

        /* 直到松开才返回 */
        fb_disp_cross(lcd_x, lcd_y, 0);
    }
```

修改 touchcreen.c 文件添加压力值相关信息

```
    //定义压力值全局变量
    static int g_ts_pressure;
    void report_ts_xy(int x, int y, int pressure);

    //发现不能画线，修改 ts_read

    //我们需要判断触摸笔是按下还是松开，需要添加压力值参数
    /*
     * 读 TS 原始数据, 转换为 LCD 坐标
```

```
 * 我们需要加上压力值 lcd_pressure
 */
int ts_read(int *lcd_x, int *lcd_y, int *lcd_pressure)
{
   int ts_x, ts_y, ts_pressure;
   int tmp_x, tmp_y;
//添加压力值参数
   ts_read_raw(&ts_x, &ts_y, &ts_pressure);

   if (g_ts_xy_swap)
   {
          swap_xy(&ts_x, &ts_y);
   }


   /* 使用公式计算 */
   tmp_x = g_kx * (ts_x - g_ts_xc) + g_lcd_xc;
   tmp_y = g_ky * (ts_y - g_ts_yc) + g_lcd_yc;

//如果值超出了 LCD 范围返回-1
   if (tmp_x < 0 || tmp_x >= xres || tmp_y < 0 ||
tmp_y >= yres)
          return -1;

   *lcd_x = tmp_x;
   *lcd_y = tmp_y;
//压力值等于全局变量 ts_pressure
   *lcd_pressure = ts_pressure;
   return 0;
}


void ts_read_raw(int *px, int *py, int *ppressure)
{
   while (g_ts_data_valid == 0);
   *px = g_ts_x;
   *py = g_ts_y;
   *ppressure = g_ts_pressure;
   g_ts_data_valid = 0;
}
```

```c
//我们需要report 上报压力值数据
void report_ts_xy(int x, int y, int pressure)
{
	//printf("x = %08d, y = %08d\n\r", x, y);

	if (g_ts_data_valid == 0)
	{
		g_ts_x = x;
		g_ts_y = y;
		g_ts_pressure = pressure;
		g_ts_data_valid = 1;
	}
}


void Isr_Tc(void)
{
	//printf("ADCUPDN = 0x%x, ADCDAT0 = 0x%x, ADCDAT1 = 0x%x, ADCTSC = 0x%x\n\r", ADCUPDN, ADCDAT0, ADCDAT1, ADCTSC);

	if (ADCDAT0 & (1<<15))
	{
		//printf("pen up\n\r");

		enter_wait_pen_down_mode();
//如果松开就上报数据xy 坐标00 压力值0
		report_ts_xy(0, 0, 0);

	}
	else
	{
		//printf("pen down\n\r");

		/* 进入"自动测量"模式 */
		enter_auto_measure_mode();

		/* 启动ADC */
```

```
            ADCCON |= (1<<0);
        }
    }


    void Isr_Adc(void)
    {
        int x = ADCDAT0;
        int y = ADCDAT1;

        static int adc_cnt = 0;
        static int adc_x = 0;
        static int adc_y = 0;


        if (!(x & (1<<15))) /* 如果仍然按下才打印 */
        {
    #if 0
            x &= 0x3ff;
            y &= 0x3ff;

            //printf("x = %08d, y = %08d\n\r", x, y);
    //当我得到触电数据以后，如果当前仍是按下状态，会上报 XY 坐标值并
且上报压力值，压力值等于 1
            report_ts_xy(x, y, 1);

            /* 启动定时器以再次读取数据 */
            ts_timer_enable();
    #endif
    //防止数据在最后出现很大的误差
            /* 第 1 次启动 ADC 后:
             *   a. 要连续启动 N 次, 获得 N 个数据, 求平均值并上报
             *   b. 得到 N 次数据后, 再启动 TIMER
             */
    //我们直接累加
            adc_x += (x & 0x3ff);
```

```
        adc_y += (y & 0x3ff);
        adc_cnt++;
//我们取 16 的话右移 4 位比较容易操作
        if (adc_cnt == 16)
        {
//右移 4 位
            adc_x >>= 4;
            adc_y >>= 4;
//上报
            report_ts_xy(adc_x, adc_y, 1);

//恢复到初始值 0
            adc_cnt = 0;
            adc_x = 0;
            adc_y = 0;

            /* 启动定时器以再次读取数据 */
            ts_timer_enable();
        }
        else
        {
            /* 否则再次启动 ADC */
            /* 进入"自动测量"模式 */
            enter_auto_measure_mode();

            /* 启动 ADC */
            ADCCON |= (1<<0);
        }

    }
    else
    {
        adc_cnt = 0;
        adc_x = 0;
        adc_y = 0;
```

```
            ts_timer_disable();
            enter_wait_pen_down_mode();
    //如果数据转换完之前再次松开，这里我也会上报数据，0，0，0
            report_ts_xy(0, 0, 0);
        }

        enter_wait_pen_up_mode();
    }
```

那么我们的 tslib.c 中

```
    void get_calibrate_point_data(int lcd_x, int lcd_y, int
*px, int *py)
    {
        int pressure;
        int x, y;

        fb_disp_cross(lcd_x, lcd_y, 0xffffff);


        /* 等待点击 */


        do {
    //读取 ts 坐标值也加上压力值
            ts_read_raw(&x, &y, &pressure);
    //直到压力等于 0 的时候才返回
        } while (pressure == 0);


        do {
            *px = x;
            *py = y;
    //压力值存在的情况下表示按下状态，就一直读取数据
            ts_read_raw(&x, &y, &pressure);
            printf("get raw data: x = %08d, y
= %08d\n\r", x, y);
        } while (pressure);
    //打印坐标值
        printf("return raw data: x = %08d, y = %08d\n\r",
*px, *py);
```

```c
    /* 直到松开才返回 */
//我们操作完成后消除 +
    fb_disp_cross(lcd_x, lcd_y, 0);
}
```

我们确定校准为什么不对，我们校准涉及 原始数据 校准公式

```c
//我们把公式使用函数来表示
int get_lcd_x_frm_ts_x(int ts_x)
{
    return g_kx * (ts_x - g_ts_xc) + g_lcd_xc;
}

int get_lcd_y_frm_ts_y(int ts_y)
{
    return g_ky * (ts_y - g_ts_yc) + g_lcd_yc;
}


void ts_calibrate(void)
{

    int a_ts_x, a_ts_y;
    int b_ts_x, b_ts_y;
    int c_ts_x, c_ts_y;
    int d_ts_x, d_ts_y;
    int e_ts_x, e_ts_y;

    /* X 轴方向 */
    int ts_s1, ts_s2;
    int lcd_s;

    /* Y 轴方向 */
    int ts_d1, ts_d2;
    int lcd_d;

    /* 获得 LCD 的参数: fb_base, xres, yres, bpp */
```

```
	get_lcd_params(&fb_base, &xres, &yres, &bpp);

	/* 对于ABCDE, 循环: 显示"+"、点击、读ts原始值 */
	/* A(50, 50) */
	get_calibrate_point_data(50, 50, &a_ts_x, &a_ts_y);

	/* B(xres-50, 50) */
	get_calibrate_point_data(xres-50, 50, &b_ts_x,
&b_ts_y);

	/* C(xres-50, yres-50) */
	get_calibrate_point_data(xres-50, yres-50, &c_ts_x,
&c_ts_y);

	/* D(50, yres-50) */
	get_calibrate_point_data(50, yres-50, &d_ts_x,
&d_ts_y);

	/* E(xres/2, yres/2) */
	get_calibrate_point_data(xres/2, yres/2, &e_ts_x,
&e_ts_y);


	/* 确定触摸屏数据XY是否反转 */

	g_ts_xy_swap = is_ts_xy_swap(a_ts_x, a_ts_y,
b_ts_x, b_ts_y);

	if (g_ts_xy_swap)
	{
		/* 对调所有点的XY坐标 */

		swap_xy(&a_ts_x, &a_ts_y);
		swap_xy(&b_ts_x, &b_ts_y);
		swap_xy(&c_ts_x, &c_ts_y);
		swap_xy(&d_ts_x, &d_ts_y);
		swap_xy(&e_ts_x, &e_ts_y);
	}

	/* 确定公式的参数并保存 */

	ts_s1 = b_ts_x - a_ts_x;
	ts_s2 = c_ts_x - d_ts_x;
	lcd_s = xres-50 - 50;
```

```
    ts_d1 = d_ts_y - a_ts_y;
    ts_d2 = c_ts_y - b_ts_y;
    lcd_d = yres-50-50;

    g_kx = ((double)(2*lcd_s)) / (ts_s1 + ts_s2);
    g_ky = ((double)(2*lcd_d)) / (ts_d1 + ts_d2);

    g_ts_xc = e_ts_x;
    g_ts_yc = e_ts_y;

    g_lcd_xc = xres/2;
    g_lcd_yc = yres/2;
//打印ABCDE的坐标值
    printf("A lcd_x = %08d, lcd_y = %08d\n\r",
get_lcd_x_frm_ts_x(a_ts_x), get_lcd_y_frm_ts_y(a_ts_y));
    printf("B lcd_x = %08d, lcd_y = %08d\n\r",
get_lcd_x_frm_ts_x(b_ts_x), get_lcd_y_frm_ts_y(b_ts_y));
    printf("C lcd_x = %08d, lcd_y = %08d\n\r",
get_lcd_x_frm_ts_x(c_ts_x), get_lcd_y_frm_ts_y(c_ts_y));
    printf("D lcd_x = %08d, lcd_y = %08d\n\r",
get_lcd_x_frm_ts_x(d_ts_x), get_lcd_y_frm_ts_y(d_ts_y));
    printf("E lcd_x = %08d, lcd_y = %08d\n\r",
get_lcd_x_frm_ts_x(e_ts_x), get_lcd_y_frm_ts_y(e_ts_y));
}
```

我们转换出的 XY 坐标值不是特别稳定

```
/* 每10ms 该函数被调用一次
 */
void touchscreen_timer_irq(void)
{
    /* 如果触摸屏仍被按下, 进入"自动测量模式", 启动ADC */
    if (get_status_of_ts_timer() == 0)
        return;

    if (ADCDAT0 & (1<<15)) /* 如果松开 */
    {
        ts_timer_disable();
        enter_wait_pen_down_mode();
        //report_ts_xy(0, 0, 0);
```

```
        return;
    }
    else  /* 按下状态 */
    {
        /* 进入"自动测量"模式 */
        enter_auto_measure_mode();

        /* 启动ADC */
        ADCCON |= (1<<0);
    }
}
```

//修改 touchscreen-test.c 文件

```
void touchscreen_test(void)
{
    unsigned int fb_base;
    int xres, yres, bpp;

    int x, y, pressure;

    /* 获得LCD的参数: fb_base, xres, yres, bpp */
    get_lcd_params(&fb_base, &xres, &yres, &bpp);

    touchscreen_init();

    /* 清屏 */
    clear_screen(0);

    /* 显示文字提示较准 */
    fb_print_string(70, 70, "Touc cross to calibrate
touchscreen", 0xffffff);
    ts_calibrate();

    /* 显示文字提示绘画 */
```

```
        fb_print_string(70, yres - 70, "OK! To draw!",
0xffffff);

        while (1)
        {
    //如果结果=0 则继续执行下面操作
                if (ts_read(&x, &y, &pressure) == 0)
                {
                        printf(" x = %d, y = %d\n\r", x, y);
    //如果是按下状态，才会描点
                        if (pressure)
                        {

                                fb_put_pixel(x, y, 0xff00);
                        }
                }
        }
    }
```

把 LCD 的电压值，成功转化成屏幕的坐标 要点

1. 对于触摸屏要多次测量，求平均值
2. 要丢弃非法值（以 LCD 分辨率作为判断标准）
3. 校准时一定要点准

参考 tslib 库，

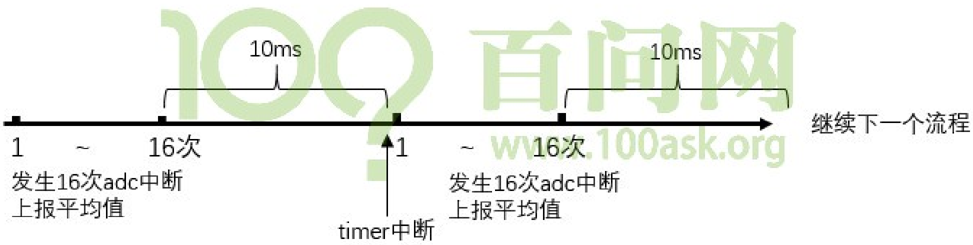
第 012 节_触摸屏编程_完善

我们触摸屏校准虽然可以正常运行，但是有些问题，比如在触摸屏上点一个点，同时屏幕上面会显示另一个点 我们按住屏幕不动的同时将其转换成 LCD 坐标并且描点，就表明数值不大稳定 问题

1. 我们第一次点击触摸屏会出现两个点
2. 长按，LCD 上的点会越来越大

根源在于我们得到的 LCD 坐标值不稳定，根源 ADC 转换出来的 xy 坐标值不稳

定



我们打开 touchscreen.c 问题出现在这里面

```
#include "../s3c2440_soc.h"

#define ADC_INT_BIT (10)
#define TC_INT_BIT  (9)

#define INT_ADC_TC   (31)


/* ADCTSC's bits */
#define WAIT_PEN_DOWN    (0<<8)
#define WAIT_PEN_UP      (1<<8)

#define YM_ENABLE        (1<<7)
#define YM_DISABLE       (0<<7)

#define YP_ENABLE        (0<<6)
#define YP_DISABLE       (1<<6)

#define XM_ENABLE        (1<<5)
#define XM_DISABLE       (0<<5)

#define XP_ENABLE        (0<<4)
#define XP_DISABLE       (1<<4)

#define PULLUP_ENABLE    (0<<3)
#define PULLUP_DISABLE   (1<<3)

#define AUTO_PST         (1<<2)

#define WAIT_INT_MODE    (3)
#define NO_OPR_MODE      (0)
```

```c
static volatile int g_ts_timer_enable = 0;

static int g_ts_x;
static int g_ts_y;
static int g_ts_pressure;
static int g_ts_data_valid = 0;


//定义测试数据16

static int test_x_array[16];
static int test_y_array[16];

void report_ts_xy(int x, int y, int pressure);

void enter_wait_pen_down_mode(void)
{
    ADCTSC = WAIT_PEN_DOWN | PULLUP_ENABLE | YM_ENABLE |
YP_DISABLE | XP_DISABLE | XM_DISABLE | WAIT_INT_MODE;
}

void enter_wait_pen_up_mode(void)
{
    ADCTSC = WAIT_PEN_UP | PULLUP_ENABLE | YM_ENABLE |
YP_DISABLE | XP_DISABLE | XM_DISABLE | WAIT_INT_MODE;
}

void enter_auto_measure_mode(void)
{
    ADCTSC = AUTO_PST | NO_OPR_MODE;
}

int is_in_auto_mode(void)
{
    return ADCTSC & AUTO_PST;
}

void Isr_Tc(void)
{
    //printf("ADCUPDN = 0x%x, ADCDAT0 = 0x%x, ADCDAT1 =
0x%x, ADCTSC = 0x%x\n\r", ADCUPDN, ADCDAT0, ADCDAT1,
ADCTSC);

    if (ADCDAT0 & (1<<15))
    {
```

```
//按下状态启用触摸屏

//启动测量模式，转换结束产生 adc 中断
        //printf("pen up\n\r");
        enter_wait_pen_down_mode();
        report_ts_xy(0, 0, 0);

    }
    else
    {
        //printf("pen down\n\r");

        /* 进入"自动测量"模式 */

        enter_auto_measure_mode();

        /* 启动 ADC */

        ADCCON |= (1<<0);
    }
}


static void ts_timer_enable(void)
{
    g_ts_timer_enable = 1;
}

static void ts_timer_disable(void)
{
    g_ts_timer_enable = 0;
}

static int get_status_of_ts_timer(void)
{
    return g_ts_timer_enable;
}


void report_ts_xy(int x, int y, int pressure)
{
    //printf("x = %08d, y = %08d\n\r", x, y);
```

```
	if (g_ts_data_valid == 0)
	{
		g_ts_x = x;
		g_ts_y = y;
		g_ts_pressure = pressure;
		g_ts_data_valid = 1;
	}
}

void ts_read_raw(int *px, int *py, int *ppressure)
{
	while (g_ts_data_valid == 0);
	*px = g_ts_x;
	*py = g_ts_y;
	*ppressure = g_ts_pressure;
	g_ts_data_valid = 0;
}


/* 每10ms 该函数被调用一次
 */
void touchscreen_timer_irq(void)
{
	/* 如果触摸屏仍被按下，进入"自动测量模式"，启动ADC */
	if (get_status_of_ts_timer() == 0)
		return;

	if (is_in_auto_mode())
		return;

	/* 只有在"等待中断模式"下才可以使用ADCDAT0'BIT 15 来判断
触摸笔状态 */

	if (ADCDAT0 & (1<<15)) /* 如果松开 */
	{
		printf("timer set pen down\n\r");
		ts_timer_disable();
		enter_wait_pen_down_mode();
```

```
        report_ts_xy(0, 0, 0);
        return;
    }
    else  /* 按下状态 */
    {
        /* 进入"自动测量"模式 */
        enter_auto_measure_mode();

        /* 启动ADC */
        ADCCON |= (1<<0);
    }
}

void Isr_Adc(void)
{
    int x = ADCDAT0;
    int y = ADCDAT1;

    static int adc_cnt = 0;
    static int adc_x = 0;
    static int adc_y = 0;

    /* 进入ADC 中断时, TS 处于"自动测量模式" */

    /* 只有在"等待中断模式"下才可以使用ADCDAT0'BIT 15 来判断
触摸笔状态 */

    enter_wait_pen_up_mode();
    if (!(ADCDAT0 & (1<<15))) /* 如果仍然按下才打印 */
    {
#if 0
        x &= 0x3ff;
        y &= 0x3ff;

        //printf("x =  %08d, y = %08d\n\r", x, y);
        report_ts_xy(x, y, 1);
```

```
            /* 启动定时器以再次读取数据 */
            ts_timer_enable();
#endif
            /* 第1次启动ADC后:
             *  a. 要连续启动N次, 获得N个数据, 求平均值并上报
             *  b. 得到N次数据后, 再启动TIMER
             */
            adc_x += (x & 0x3ff);
            adc_y += (y & 0x3ff);
//定义一个函数把这些值打印出来
            test_x_array[adc_cnt] = (x & 0x3ff);
            test_y_array[adc_cnt] = (y & 0x3ff);

            adc_cnt++;

            if (adc_cnt == 16)
            {
                adc_x >>= 4;
                adc_y >>= 4;
                report_ts_xy(adc_x, adc_y, 1);

                adc_cnt = 0;
                adc_x = 0;
                adc_y = 0;

                /* 启动定时器以再次读取数据 */
                /* 先设置TS进入"等待中断模式" */
//有按下就会有松开
                enter_wait_pen_up_mode();
                ts_timer_enable();
            }
            else
            {
                /* 再次启动ADC */
                /* 进入"自动测量"模式 */
```

```
                enter_auto_measure_mode();

                /* 启动ADC */
                ADCCON |= (1<<0);
            }

        }
        else
        {
            adc_cnt = 0;
            adc_x = 0;
            adc_y = 0;
            printf("adc report pen down\n\r");
            ts_timer_disable();
            enter_wait_pen_down_mode();
            report_ts_xy(0, 0, 0);
        }

        //enter_wait_pen_up_mode();  /* 启动ADC时不应该进入这
个模式, 它会影响数据 */
    }

    void AdcTsIntHandle(int irq)
    {
        if (SUBSRCPND & (1<<TC_INT_BIT))  /* 如果是触摸屏中断
*/
            Isr_Tc();

        if (SUBSRCPND & (1<<ADC_INT_BIT))  /* ADC中断 */
            Isr_Adc();
        SUBSRCPND = (1<<TC_INT_BIT) | (1<<ADC_INT_BIT);
    }

    void adc_ts_int_init(void)
    {
        SUBSRCPND = (1<<TC_INT_BIT) | (1<<ADC_INT_BIT);

        /* 注册中断处理函数 */
```

```
    register_irq(31, AdcTsIntHandle);

    /* 使能中断 */
    INTSUBMSK &= ~((1<<ADC_INT_BIT) | (1<<TC_INT_BIT));
    //INTMSK    &= ~(1<<INT_ADC_TC);
}


void adc_ts_reg_init(void)
{
    /* [15] : ECFLG,  1 = End of A/D conversion
     * [14] : PRSCEN, 1 = A/D converter prescaler
enable
     * [13:6]: PRSCVL, adc clk = PCLK / (PRSCVL + 1)
     * [5:3] : SEL_MUX, 000 = AIN 0
     * [2]   : STDBM
     * [0]   : 1 = A/D conversion starts and this bit
is cleared after the startup.
     */
    ADCCON = (1<<14) | (49<<6) | (0<<3);

    /*  按下触摸屏, 延时一会再发出 TC 中断
     *  延时时间 = ADCDLY * 晶振周期 = ADCDLY * 1 /
12000000 = 5ms
     */
    ADCDLY = 60000;
}


void touchscreen_init(void)
{
    /* 设置触摸屏接口:寄存器 */
    adc_ts_reg_init();

    printf("ADCUPDN = 0x%x, SUBSRCPND = 0x%x, SRCPND =
0x%x\n\r", ADCUPDN, SUBSRCPND, SRCPND);

    /* 设置中断 */
    adc_ts_int_init();
```

```
        /* 注册定时器处理函数 */
        register_timer("touchscreen",
touchscreen_timer_irq);

        /* 让触摸屏控制器进入"等待中断模式" */
        enter_wait_pen_down_mode();
    }


    //打印数组定义的那些值
    void print_test_array(void)
    {
        int i;

        printf("test array x : ");
        for (i = 0; i < 16; i++)
                printf("%08d ", test_x_array[i]);
        printf("\n\r");

        printf("test array y : ");
        for (i = 0; i < 16; i++)
                printf("%08d ", test_y_array[i]);
        printf("\n\r");
    }


    void ts_read_raw_test(int *px, int *py, int *ppressure)
    {
        while (g_ts_data_valid == 0);
        *px = g_ts_x;
        *py = g_ts_y;
        *ppressure = g_ts_pressure;
        print_test_array();
        g_ts_data_valid = 0;
    }
```

tslib.c 使用了 ts_read_raw 函数

```
    void get_calibrate_point_data(int lcd_x, int lcd_y, int
*px, int *py)
```

```
    {
        int pressure;
        int x, y;
        int sum_x = 0, sum_y = 0;
        int cnt = 0;

        fb_disp_cross(lcd_x, lcd_y, 0xffffff);


        /* 等待点击 */


        do {
                //ts_read_raw(&x, &y, &pressure);
    //我们使用测试程序去读这些值
                ts_read_raw_test(&x, &y, &pressure);
        } while (pressure == 0);


        do {
                if (cnt < 128)
                {
                        sum_x += x;
                        sum_y += y;
                        cnt++;
                }
                //ts_read_raw(&x, &y, &pressure);
    //
                ts_read_raw_test(&x, &y, &pressure);
                printf("get raw data: x = %08d, y = %08d,
cnt = %d\n\r", x, y, cnt);
        } while (pressure);

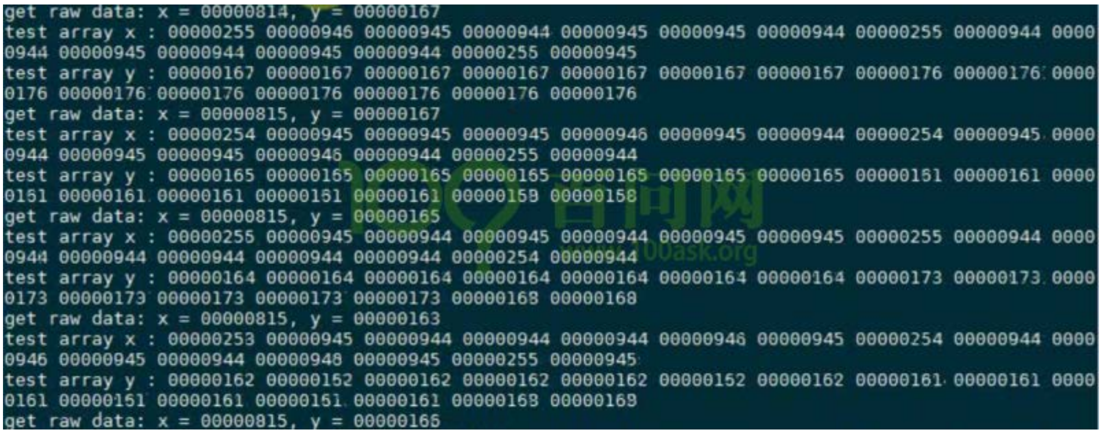
        *px = sum_x / cnt;
        *py = sum_y / cnt;

        printf("return raw data: x = %08d, y = %08d\n\r",
*px, *py);


        /* 直到松开才返回 */

        fb_disp_cross(lcd_x, lcd_y, 0);
    }
```

我们来看点击一下是不是得到了距离非常远的两个值



对于同一个点得到的是 255 945 945 944

发现 945 944 经常出现

我们查一下原因 进入 touchscreen.c 中

```
void Isr_Adc(void)
{
    int x = ADCDAT0;
    int y = ADCDAT1;

    static int adc_cnt = 0;
    static int adc_x = 0;
    static int adc_y = 0;

    /* 进入 ADC 中断时, TS 处于"自动测量模式" */

    /* 只有在"等待中断模式"下才可以使用 ADCDAT0'BIT 15 来判断
触摸笔状态 */

    enter_wait_pen_up_mode();
    if (!(ADCDAT0 & (1<<15))) /* 如果仍然按下才打印 */
    {
#if 0
            x &= 0x3ff;
            y &= 0x3ff;

            //printf("x = %08d, y = %08d\n\r", x, y);
            report_ts_xy(x, y, 1);
```

```
        /* 启动定时器以再次读取数据 */
        ts_timer_enable();
#endif
        /* 第1次启动ADC后:
         *   a. 要连续启动N次, 获得N个数据, 求平均值并上报
         *   b. 得到N次数据后, 再启动TIMER
         */
        adc_x += (x & 0x3ff);
        adc_y += (y & 0x3ff);

        test_x_array[adc_cnt] = (x & 0x3ff);
        test_y_array[adc_cnt] = (y & 0x3ff);

        adc_cnt++;

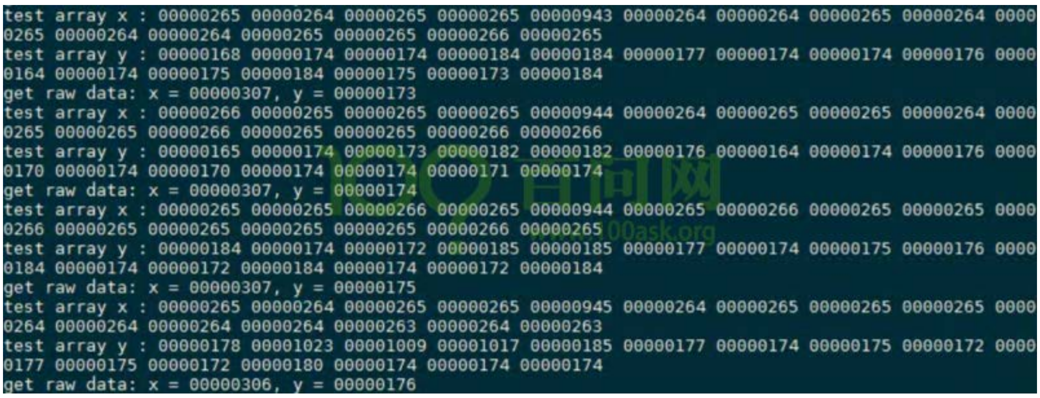
        if (adc_cnt == 16)
        {
                adc_x >>= 4;
                adc_y >>= 4;
                report_ts_xy(adc_x, adc_y, 1);

                adc_cnt = 0;
                adc_x = 0;
                adc_y = 0;

                /* 启动定时器以再次读取数据 */
                /* 先设置TS进入"等待中断模式" */
                enter_wait_pen_up_mode();
                ts_timer_enable();
        }
        else
        {
                /* 再次启动ADC */
                /* 进入"自动测量"模式 */
                enter_auto_measure_mode();
```

```
                /* 启动ADC */
                ADCCON |= (1<<0);
            }

        }
        else
        {
            adc_cnt = 0;
            adc_x = 0;
            adc_y = 0;
            printf("adc report pen down\n\r");
            ts_timer_disable();
            enter_wait_pen_down_mode();
            report_ts_xy(0, 0, 0);
        }
    //启动ADC后又再次进入enter_wait_pen_up_mode电阻上拉，多次一举
        //enter_wait_pen_up_mode();  /* 启动ADC时不应该进入这个模式，它会影响数据 */
    }
```



```
test array x : 00000265 00000264 00000265 00000265 00000943 00000264 00000264 00000265 00000264 0000
0265 00000264 00000264 00000265 00000265 00000266 00000265
test array y : 00000168 00000174 00000174 00000184 00000184 00000177 00000174 00000174 00000176 0000
0164 00000174 00000175 00000184 00000175 00000173 00000184
get raw data: x = 00000307, y = 00000173
test array x : 00000266 00000265 00000265 00000265 00000944 00000264 00000265 00000265 00000264 0000
0265 00000265 00000266 00000265 00000265 00000266 00000266
test array y : 00000165 00000174 00000173 00000182 00000182 00000176 00000164 00000174 00000176 0000
0170 00000174 00000170 00000174 00000174 00000171 00000174
get raw data: x = 00000307, y = 00000174
test array x : 00000265 00000265 00000266 00000265 00000944 00000265 00000266 00000265 00000265 0000
0266 00000265 00000265 00000265 00000265 00000266 00000265
test array y : 00000184 00000174 00000172 00000185 00000185 00000177 00000174 00000175 00000176 0000
0184 00000174 00000172 00000184 00000174 00000172 00000184
get raw data: x = 00000307, y = 00000175
test array x : 00000265 00000264 00000265 00000265 00000945 00000264 00000265 00000265 00000265 0000
0264 00000264 00000264 00000264 00000263 00000264 00000263
test array y : 00000178 00001023 00001009 00001017 00000185 00000177 00000174 00000175 00000172 0000
0177 00000175 00000172 00000180 00000174 00000174 00000174
get raw data: x = 00000306, y = 00000176
```

发现这些值中还有 944，我需要继续查找原因 在 touchscreen.c 时钟处理函数中添加打印信息

```
/* 每10ms该函数被调用一次
 */
void touchscreen_timer_irq(void)
{
    /* 如果触摸屏仍被按下，进入"自动测量模式"，启动ADC */
```

```c
        if (get_status_of_ts_timer() == 0)
                return;

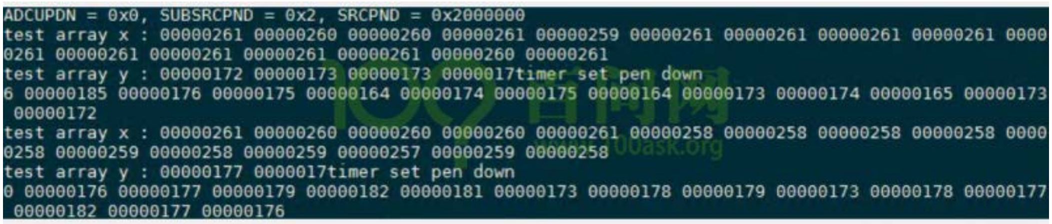
        if (is_in_auto_mode())
                return;

        /* 只有在"等待中断模式"下才可以使用ADCDAT0'BIT 15来判断
触摸笔状态 */

        if (ADCDAT0 & (1<<15)) /* 如果松开 */
        {
    //添加打印信息
                printf("timer set pen down\n\r");
                ts_timer_disable();
    //进入enter_wait_pen_down_mode，来判断触摸笔是按下还是松开
                enter_wait_pen_down_mode();
                report_ts_xy(0, 0, 0);
                return;
        }

        else  /* 按下状态 */
        {
                /* 进入"自动测量"模式 */
                enter_auto_measure_mode();

                /* 启动ADC */
                ADCCON |= (1<<0);
        }
    }
```



非常频繁打印 timer set pen down
我们的判断有问题
打开芯片手册，搜索这个寄存器 Bit15 确实是判断按下或者松开

ADC CONVERSION DATA REGISTER (ADCDAT0)

Register	Address	R/W	Description	Reset Value
ADCDAT0	0x580000C	R	ADC conversion data register	–

ADCDAT0	Bit	Description	Initial State
UPDOWN	[15]	Up or Down state of stylus at waiting for interrupt mode. 0 = Stylus down state. 1 = Stylus up state.	–

只有在中断模式下，这一位才可以正确反应是按下还是松开的状态 修改 touchscreen_timer_irq 函数

```
/* 每10ms 该函数被调用一次
 */
void touchscreen_timer_irq(void)
{
    /* 如果触摸屏仍被按下, 进入"自动测量模式", 启动ADC */
    if (get_status_of_ts_timer() == 0)
        return;

//如果定时器中断在ADC 中间产生, 应该立刻返回啥都不做, 不需要
timer 去做任何操作
//我们需要写出这个函数
//如果不是自动模式, 那么就是等待中断模式
    if (is_in_auto_mode())
        return;

//中断处理函数,
    /* 只有在"等待中断模式"下才可以使用ADCDAT0'BIT 15 来判断
触摸笔状态 */

    if (ADCDAT0 & (1<<15)) /* 如果松开 */
```

```
    {
            printf("timer set pen down\n\r");
            ts_timer_disable();
            enter_wait_pen_down_mode();
            report_ts_xy(0, 0, 0);
            return;
    }
    else  /* 按下状态 */
    {
            /* 进入"自动测量"模式 */
            enter_auto_measure_mode();

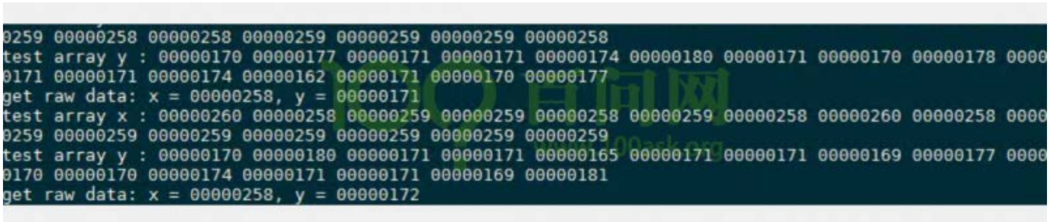
            /* 启动ADC */
            ADCCON |= (1<<0);
    }
}


//判断是否是模式模式的函数

int is_in_auto_mode(void)
{

    return ADCTSC & AUTO_PST;
}
```

我们接着实验


```
0259 00000258 00000258 00000259 00000259 00000259 00000258
test array y : 00000170 00000177 00000171 00000171 00000174 00000180 00000171 00000170 00000178 0000
0171 00000171 00000174 00000162 00000171 00000170 00000177
get raw data: x = 00000258, y = 00000171
test array x : 00000260 00000258 00000259 00000259 00000258 00000259 00000258 00000260 00000258 0000
0259 00000259 00000259 00000259 00000259 00000259 00000259
test array y : 00000170 00000180 00000171 00000171 00000165 00000171 00000171 00000169 00000177 0000
0170 00000170 00000174 00000171 00000171 00000169 00000181
get raw data: x = 00000258, y = 00000172
```

发现并没有捕捉到笔的松开模式
修改 tslib，取消 ts_read_raw_test 重新进行测试

```
void get_calibrate_point_data(int lcd_x, int lcd_y, int
*px, int *py)
{
    int pressure;
    int x, y;
```

```c
        int sum_x = 0, sum_y = 0;
        int cnt = 0;

        fb_disp_cross(lcd_x, lcd_y, 0xffffff);


        /* 等待点击 */


        do {
                ts_read_raw(&x, &y, &pressure);
                //ts_read_raw_test(&x, &y, &pressure);
        } while (pressure == 0);


        do {
                if (cnt < 128)
                {
                        sum_x += x;
                        sum_y += y;
                        cnt++;
                }
                ts_read_raw(&x, &y, &pressure);
                //ts_read_raw_test(&x, &y, &pressure);
                printf("get raw data: x = %08d, y = %08d, 
cnt = %d\n\r", x, y, cnt);
        } while (pressure);

        *px = sum_x / cnt;
        *py = sum_y / cnt;

        printf("return raw data: x = %08d, y = %08d\n\r", 
*px, *py);


        /* 直到松开才返回 */

        fb_disp_cross(lcd_x, lcd_y, 0);
    }
```

//现在发现点点不准确



发现校准的值和我们之前的不一样
修改我们的 tslib 校准程序

```
void get_calibrate_point_data(int lcd_x, int lcd_y, int
*px, int *py)
{
    int pressure;
    int x, y;
    int sum_x = 0, sum_y = 0;
    int cnt = 0;

    fb_disp_cross(lcd_x, lcd_y, 0xffffff);


    /* 等待点击 */


    do {
            ts_read_raw(&x, &y, &pressure);
            //ts_read_raw_test(&x, &y, &pressure);
    } while (pressure == 0);


    do {
//我们把求和的次数限制为 128 次
            if (cnt < 128)
            {
                    sum_x += x;
                    sum_y += y;
                    cnt++;
            }
            ts_read_raw(&x, &y, &pressure);
            //ts_read_raw_test(&x, &y, &pressure);
            printf("get raw data: x = %08d, y = %08d,
cnt = %d\n\r", x, y, cnt);
    } while (pressure);

    *px = sum_x / cnt;
    *py = sum_y / cnt;

    printf("return raw data: x = %08d, y = %08d\n\r",
*px, *py);


    /* 直到松开才返回 */
```

```
    fb_disp_cross(lcd_x, lcd_y, 0);
}
```

我们可以参考 tslib

1. 使用矩阵进行校准，适用性更强
2. 使用多种方法消除误差，多次测量求平均值

判断相领点的距离，如果突然变化很大，就有可能是错误值

1. ...

第一期的视频在于裸机基本操作 视频的要点在于
修改要点

1. 启动 ADC 时不应该进入等待中断模式，它会影响数据
2. 只有在"等待中断模式"下才可以使用 ADCDAT0'BIT 15 来判断触摸笔状态
3. 校准非常重要，所以在程序种多次测量求平均值(不仅仅是在 adc 中断种求平均值)

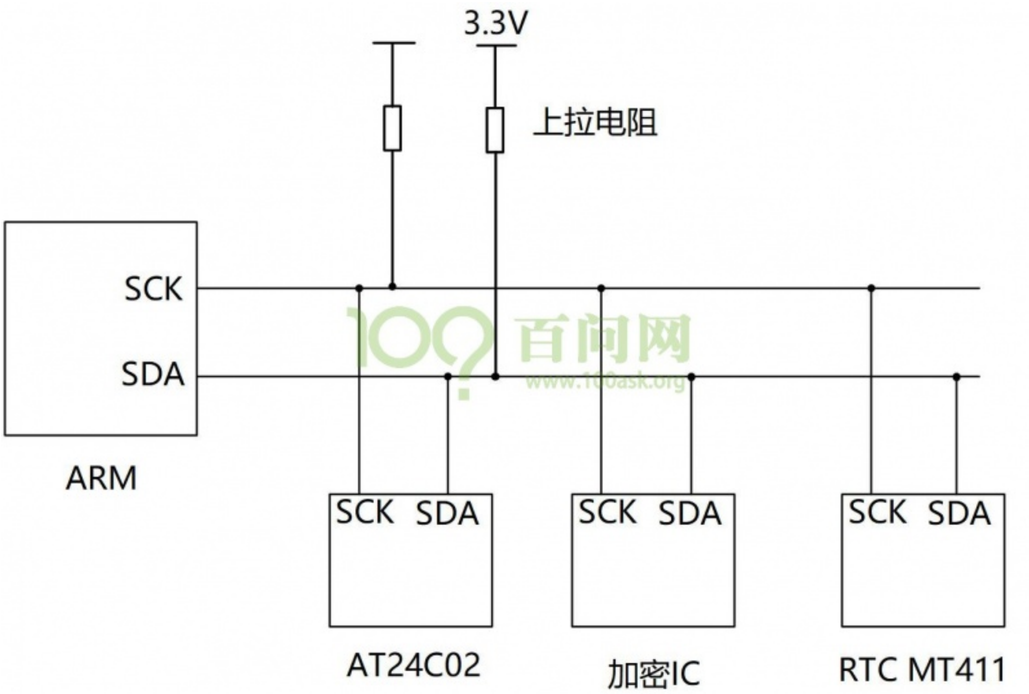
第 019 课 I2C

第 001 节_I2C 协议与 EEPROM

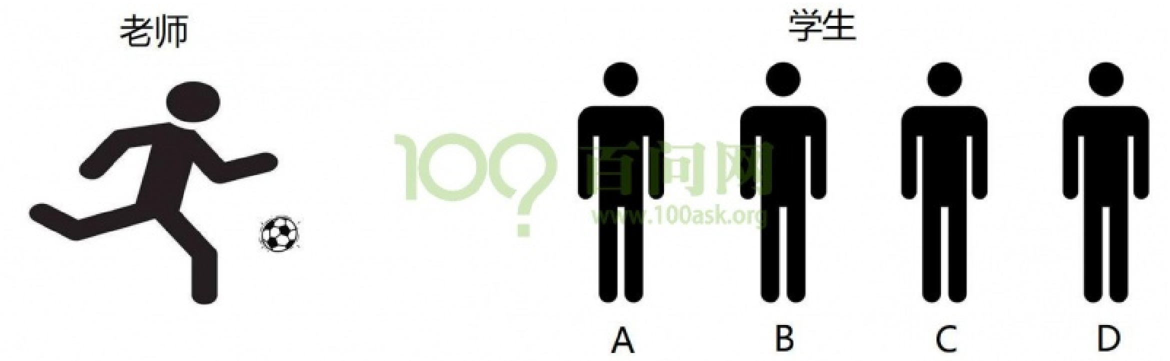
I2C 协议

I2C 在硬件上的接法如下(图 19-1)所示，主控芯片引出两条线 SCL,SDA 线，在一条 I2C 总线上可以接很多 I2C 设备，我们还会放一个上拉电阻（放一个上拉电阻的原因以后我们再

说）。



我们怎么传输数据，我们需要发数据从主设备发送到从设备上去，也需要把数据从从设备传送到主设备上去，数据涉及到双向传输。举个例子：



体育老师：可以把球发给学生，也可以把球从学生中接过来。 1，发球： a，老师说：注意了(start) b，老师对 A 学生说我要球发给你(地址)。 c，老师就把球发出去了（传输）。 d,A 收到球之后，应该告诉老师一声（回应）。 e,老师说下课（停止）

2，接球： a，老师说注意了(start)， b，老师说：B 把球发给我(地址) c,B 就把球发给老师（传输） d,老师收到球之后，给 B 说一声，表示收到球了（回应）。 e,老师说下课（停止）

我们就使用这个简单的例子，来解释一下 IIC 的传输协议。 老师说注意了，表示开始信号(start) 老师告诉某个学生，表示发送地址(address) 老师发球/接球，表示数据的传输 老师/学生收到球，回应表示：回应信号(ACK) 老师说下课，表示 IIC 传输接受(P)

IIC 传输数据的格式

1，写操作： 刚开始主芯片要发出一个 start 信号，然后发出一个设备地址(用来确定是往哪一个芯片写数据)，方向(读/写，0 表示写，1 表示读)。 回应

(用来确定这个设备是否存在)，然后就可以传输数据，传输数据之后，要有一个回应信号（确定数据是否接受完成)，然后再传输下一个数据。 每传输一个数据，接受方都会有一个回应信号，数据发送完之后，主芯片就会发送一个停止信号。 白色背景：主→从灰色背景：从→主



2，读操作： 刚开始主芯片要发出一个 start 信号，然后发出一个设备地址(用来确定是从哪一个芯片读取数据)，方向(读/写，0 表示写，1 表示读)。 回应(用来确定这个设备是否存在)，然后就可以传输数据，传输数据之后，要有一个回应信号（确定数据是否接受完成)，然后在传输下一个数据。 每传输一个数据，接受方都会有一个回应信号，数据发送完之后，主芯片就会发送一个停止信号。 白色背景：主→从灰色背景：从→主



传输是以 8 位为单元数据传输的，先传输最高位(MSB)，主芯片发出 start 信号之后，然后发出 9 个时钟传输数据。 （1）开始信号（S)：SCL 为高电平时，SDA 山高电平向低电平跳变，开始传送数据。 （2）结束信号（P)：SCL 为电平时，sDA 由低电平向高电平跳变，结束传送数据。 （3）响应信号(ACK)：接收器在接收到 8 位数据后，在第 9 个时钟周期，拉低 SDA SDA 上传输的数据必须在 SCL 为高电平期间保持稳定，SDA 上的数据只能在 SCL 为低电平期间变化。如

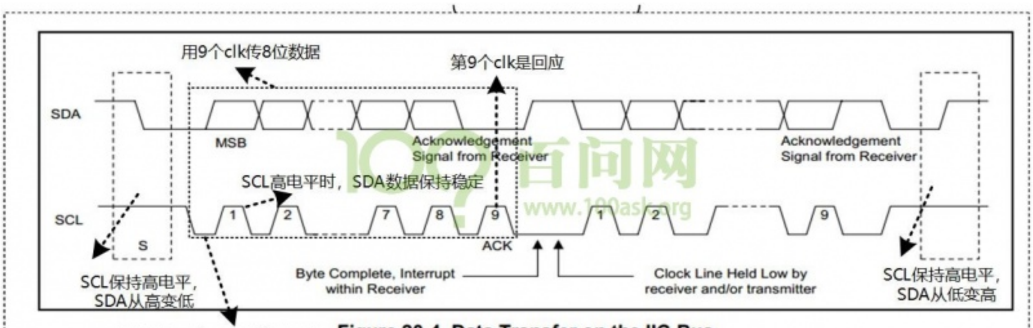
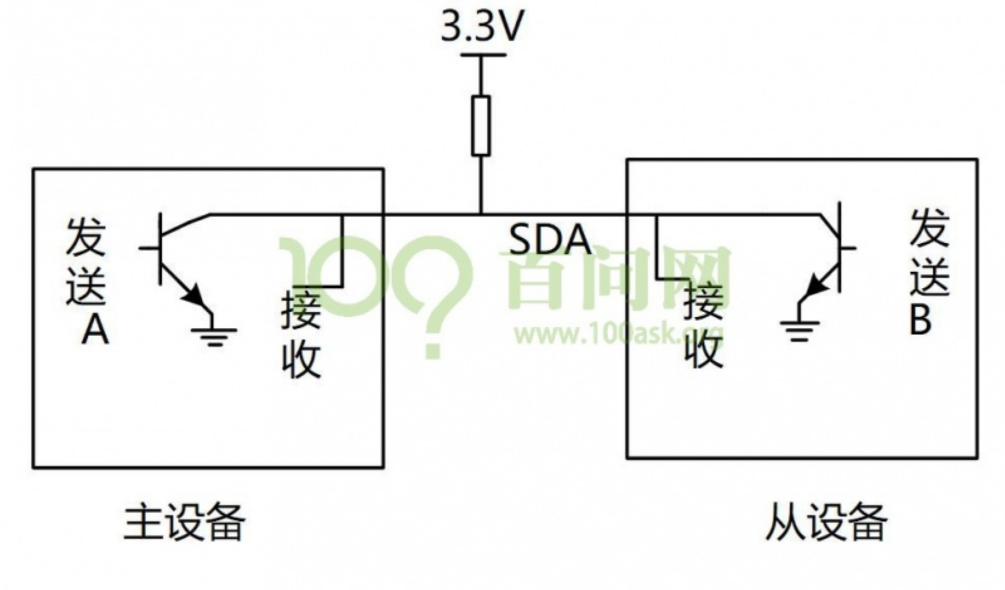


图 Figure 20-4. Data Transfer on the IIC-Bus

1，问题：如何在 SDA 上实现双向传输？ 答：主芯片通过一根 SDA 线既可以把数据发给从设备，也可以从 SDA 上读取数据，连接 SDA 线的引脚里面必然有两个引脚（发送引脚/接受引脚)。

2，问题：主设备(从设备)发送数据时，从设备(主设备)的发送引脚，不影响数据的发送，怎么做到呢？ 答：里面放一个三极管，使用开极(极电集开发

出去作为输出)电路，如下



图

下面画一个真值

A	B	SDA
0	0	1(由上拉电阻决定)
0	1	0
1	0	0
1	1	0

表:

从真值表和电路图我们可以知道，当某一个芯片不行影响 SDA 线时，那就不驱动这个三极管。

想输出高电平时；都不驱动(高电平就由上拉电阻决定)。

想输出低电平，就驱动三极管。

从下面的例子可以看看数据是怎么传的（实现双向传输），比如：主设备发送（8bit）给从设备 1， 前 8 个 clk

从设备不要影响，从设备不驱动三极管；

主设备决定数据；

2， 第 9 个 clk，由从设备决定数据

主设备不驱动三极管；

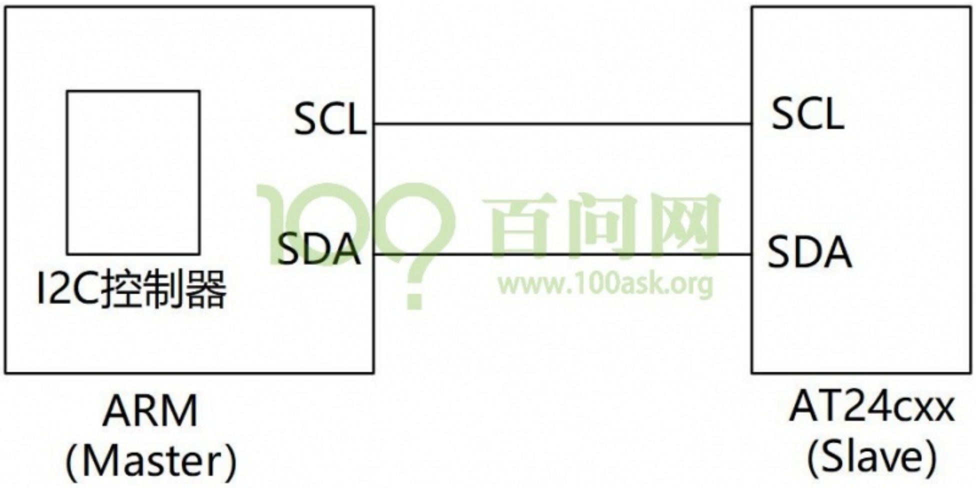
从设备决定数据；

从上面的例子，就可以知道，怎样在一条线上实现，双向传输的办法。这就是为什么在 SDA,SCL 上放上拉电阻的原因。

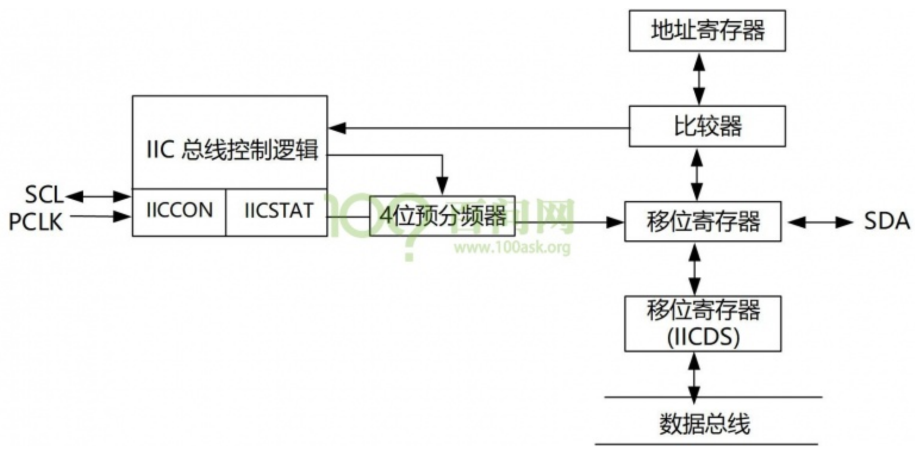
在第 9 个时钟之后，如果有某一方处于繁忙状态，它可以一直把 SCL 拉低当 SCL 为低电平时候，大家都不应该使用 IIC 总线，只有当 SCL 从低电平变为高电平的时候，IIC 总线才能被使用。 从图 1-6 和图 1-7 我们也可以知道 ACK 信号应该是低电平。主设备不驱动 三极管，如果从设备不驱动三极端的化 SDA 应该是高电平，当从设备接收数据之后，发出回应信号的时候，就会驱动三极管，让 SDA 变为低电平。所以说：ACK 信号是低电平。 对于 IIC 协议它只能规定怎么传输数据，数据什么含义它完全不能够控制，数据的含义有从设备决定。

第 002 节_S3C2440 的 I2C 控制器

在嵌入式系统里面的主控芯片一般都会有 I2C 控制器，要是没有可以根据 I2C 协议用 GPIO 管脚模拟，但是非常麻烦，我们要发送数据时，可以把数据放到某个寄存器，它就会自动的发出时钟，并且把数据发送给从设备，同时会等待从设备会返回回应信号。当我们想发送一个数据的时候，要设置某个寄存器启动传输，它也一样会产生时钟，然后从设备就会把数据通过 SDA 传到 I2C 控制器里面，组装进某个寄存器里面，最终寄存器会把接收到的 8 位数据返回给我们的程序，从这里可以看到 I2C 控制器简化了 I2C 的操作。简短电路连接图，如



图：



根据上图，我们首先设置 IICCON(来设置时钟)，时钟源是 PCLK(是 50MHZ)太快了我们需要设置这个分频系数，把时钟降低，降低到我们想要的 SCL,然后我们要发

出 start 信号，我们需要设置寄存器发出 start 信号，之后我们需要发出数据啊，我们的程序可以把数据写入到 IICDS 寄存器，一写入就会自动的发出时钟，并且把这 8 位数据从 SDA 发送给从设备，数据发送之后，在第九个时钟会收到回应信号，可以查询 IICSTAT 是否有 ACK(有 ACK 表示数据发送成功了)，可以继续发送数据，等发完数据之后，再来设置 IICSTAT 让它发出 P 信号。 在第九个 CLK，就会产生一个中断,在中断处理过程中 SCL 被拉为低电平，谁都不能再使用 IIC 总线，等待中断处理完成.

怎样处理中断？

- 写操作：

若无 ACK,出错，然后发出 P 信号结束， 若有 ACK 信号表示上一个字节成功发送出去 若仍有数据，写入 IICDS 寄存器，然后清中断，一清中断就会释放 SCL 信号，继续发出时钟，把数据再次发送出去。 若没有数据了，发出 P 信号结束。

- 读操作：

读到 8 位数时，应该回应一个 ACK 信号。 还想读数据，清中断，启动传输。等它再次发生中断时，再来读取 IICDS 寄存器,得到数据。不想读取数据，发出 P 信号结束。

重点： 发生中断时，我们的 IIC 控制器会把 SCL 拉低，阻止任何设备再使用 IIC 总线，清中断之后才能继续使用，这种机制就给我们中断服务程序的执行提供了时间。

读-写操作

- 在发送模式：

1，往寄存器 IICDS 寄存器放入一个 val 值。 2，发完，产生中断，并且会把 SCL 拉低。 3，在中断程序里，判断状态，然后往 IICDS 里面写入下一个数据，一旦写入下一个数据 IIC 继续操作，若再次发完，就会再次产生中断。

- 在接受模式：

1，我的程序发起传输，接受数据。 2，接收到数据之后，产生中断，SCL 被拉低。 3，中断程序里，判断数据是否要继续接受等，如果还有继续接受的话，再次设置，设置好之后读 IICDS 寄存器，一但读出来 IIC。 继续接受下一个数据，收到新数据之后，又会产生一个中断（就是这样循环操作）。

(1)IICCON 寄存器（Multi-masterIIC-buscontrol） IICCON 寄存器用于控制是否发出 ACK 信号、设置发送器的时钟、开启，i2c 中断，并标识中断是

否发生。它的各位含义如

寄存器	地址	R/W	描述	复位值
IICCON	0x54000000	R/W	IIC 总线控制寄存器	0x0x

IICCON	位	描述	初始状态
应答发生	[7]	IIC 总线应答使能位 0 = 禁止　1 = 允许 Tx 模式中，IICSDA在应答时间为空闲。 Rx 模式中，IICSDA 在应答时间为低。	0
Tx 时钟源选择	[6]	IIC 总线发送时钟预分频器的时钟源选择位 0：IICCLK = fPCLK /16　1：IICCLK = fPCLK/512	0
Tx/Rx 中断	[5]	IIC 总线 Tx/Rx 中断使能/ 禁止位 0 = 禁止　1 = 允许	0
中断挂起标志	[4]	IIC 总线 Tx/Rx 中断挂起标志。不能写 1 到此位。当此位读取到 1 时，IICSCL 限制为低并且停止 IIC。清除此位为 0 以继续操作。 0：1）无中断挂起（读时） 2）清除挂起条件并且继续操作（写时） 1：1）中断挂起（读时） 2）N/A（写时）	0
发送时钟值	[3:0]	IIC 总线发送时钟预分频器。 IIC 总线发送时钟预分频器是由此 4 位预分频值按以下公式决定的： Tx 时钟 = IICCLK / (IICCON[3:0] + 1)	0

表:

使用 IICCON 寄存器时，有如一卜注意事项。 1,发送模式的时钟频率由位[6]、位[3:0]联合决定,另外，11CCON[6]=0,IICCON[3:0] 不能取 0 或 10 2,12c 中断在以下 3 种情况下发生：当发出地址信息或接收到一个从机地址并且吻合时,当总线仲裁失败时，当发送/接收完一个字节的数据（包括响应位）时。 3,基于 SDA、SCL 线上时间特性的考虑，要发送数据时，先将数据写入 IICDS 寄存器，然后再清除中断。 4,如果 IICCON[5]=0，IICCON14]将不能正常工作。所以，即使不使用 12c 中断，也要将 IICCON[5]设为 1。

(2)IICSTAT 寄存器(Multi-masterIIC-buscontrol/status) IICSTAT 寄存器用于选择 12c 接口的工作模式，发出 S 信号、P 信号，使能接收 / 发送功能，并标识各种状态，比如总线仲裁是否成功、作为从机时是否被寻址、是否

接收到 0 地址、是否接收到 ACK 信号等。IICSTAT 寄存器的各位如

寄存器	地址	R/W	描述	复位值
IICSTAT	0x54000004	R/W	IIC 总线控制/状态寄存器	0x0

IICSTAT	位	描述	初始状态
模式选择	[7:6]	IIC 总线主机/从机 Tx/Rx 模式选择位 00：从接收模式 01：从发送模式 10：主接收模式 11：主发送模式	00
忙信号状态/起始停止条件	[5]	IIC 总线忙信号状态位 0：读）不忙（读时） 写）停止信号产生 1：读）忙（读时） 写）起始信号产生	0
串行输出	[4]	IIC 总线 Tx/Rx 中断使能/禁止位 0 = 禁止 1 = 允许	0
仲裁状态标志	[3]	IIC 总线仲裁过程状态标志位 0：总线仲裁成功 1：串行 I/O 间总线仲裁失败	0
从地址状态标志	[2]	IIC 总线从地址状态标志位 0：发现起始/停止条件清除 1：收到从地址与 IICADD 中地址值匹配	0
地址零状态标志	[1]	IIC 总线地址零状态标志位 0：发现起始/停止条件清除 1：收到从地址为 00000000b	0
最后收到位状态标志	[0]	IIC 总线最后收到位状态标志位 0：最后收到位为 0 （已收到 ACK） 1：最后收到位为 1 （未收到 ACK）	0

表：

（3）**IICADD 寄存器(Multi-masterIlC-busaddress)** 用到 IICADD 寄存器的位[7:11]，表示从机地址。IICADD 寄存器在串行输出使能位 IICSTAT[4]为 0 时，才可以写入：在任何时间都可以读出。IICADD 寄存器的各位如

表:

寄存器	地址	R/W	描述	复位值
IICADD	0x54000008	R/W	IIC 总线地址寄存器	0x0XX

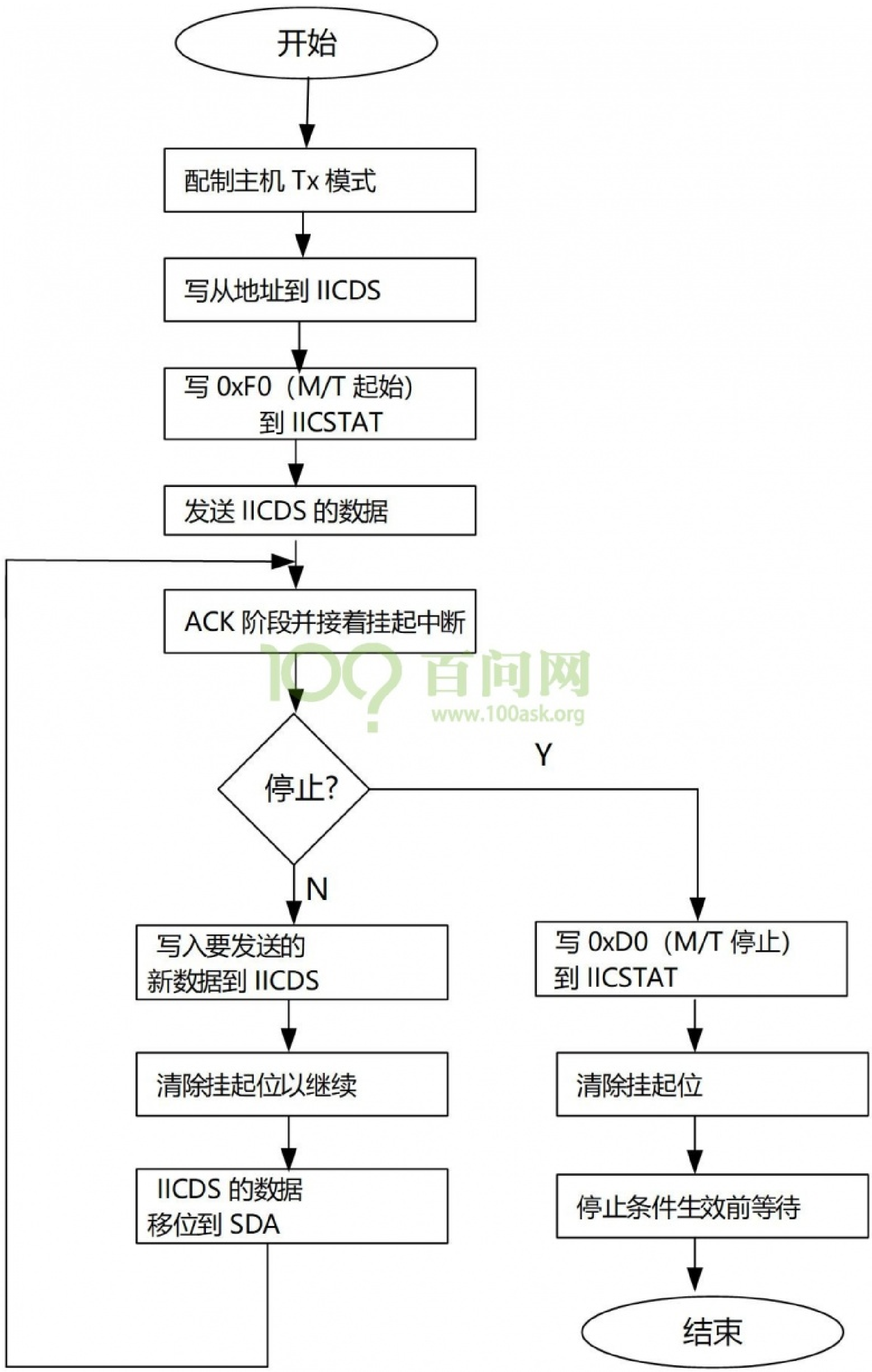
IICADD	位	描述	复位值
从地址	[7:0]	从 IIC 总线锁存的 7 位从地址。 当 IICSTAT 中串行输出使能=0 时 IICADD 为写使能。可以在任意时间读取 IICADD 的值，不用去考虑当前输出使能位（IICSTAT）的设置。 从地址：[7:1] 未映射：[0]	XXXXXXXX

（4）IICDS 寄存器(Multi-masterIIC-busTx/Rxdatashift) 用到 IICDS 寄存器的位丨 7：0]，其中保存的是要发送或已经接收的数据。IICDS 寄存器在串行输出使能位 IICSTAT()1 为 1 时，叼可以写入；在任何时间都可以读出。IICDS 寄存器的各位如表:

寄存器	地址	R/W	描述	复位值
IICDS	0x5400000C	R/W	IIC 总线发送/接收数据移位寄存器	0x0XX

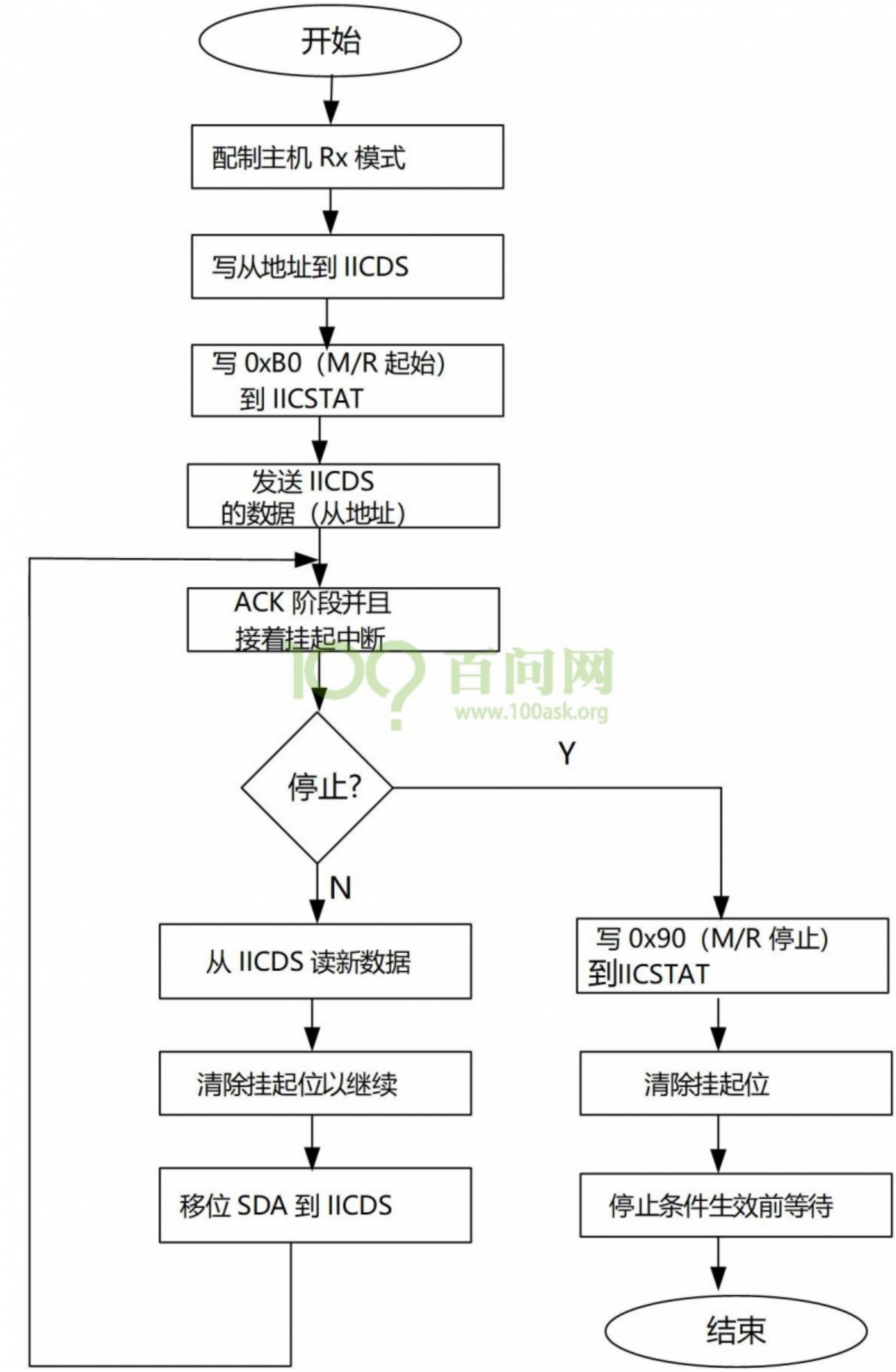
IICDS	位	描述	复位值
数据移位	[7:0]	IIC 总线 Tx/Rx 操作的 8 位数据移位寄存器。 当 IICSTAT 中串行输出使能=1，IICDS 为写使能。可以在任意时间读取 IICDS 的值，不用去考虑当前输出使能位（IICSTAT）的设置。	XXXXXXXX

读写操作流程图 主机发送器模式操作:


开始
配制主机 Tx 模式
写从地址到 IICDS
写 0xF0（M/T 起始）
到 IICSTAT
发送 IICDS 的数据
ACK 阶段并接着挂起中断
停止?
Y
N
写入要发送的
新数据到 IICDS
清除挂起位以继续
IICDS 的数据
移位到 SDA
写 0xD0（M/T 停止）
到 IICSTAT
清除挂起位
停止条件生效前等待
结束

主机接收器模式操

作:


开始
配制主机 Rx 模式
写从地址到 IICDS
写 0xB0（M/R 起始）
到 IICSTAT
发送 IICDS
的数据（从地址）
ACK 阶段并且
接着挂起中断
停止?
Y
N
从 IICDS 读新数据
清除挂起位以继续
移位 SDA 到 IICDS
写 0x90（M/R 停止）
到IICSTAT
清除挂起位
停止条件生效前等待
结束

第 003 节_程序框架

写程序之前 考虑好程序的框架，我们想写出一个结构比较好，比较容易扩展的程序 我们先要考虑清楚框架的设计。

IIC 控制器的功能

IIC 会做什么事情呢？ 对于 IIC 控制器，它负责传输数据，不知道数据的含义，但是它要实现写/读操作

读操作



'

写操作

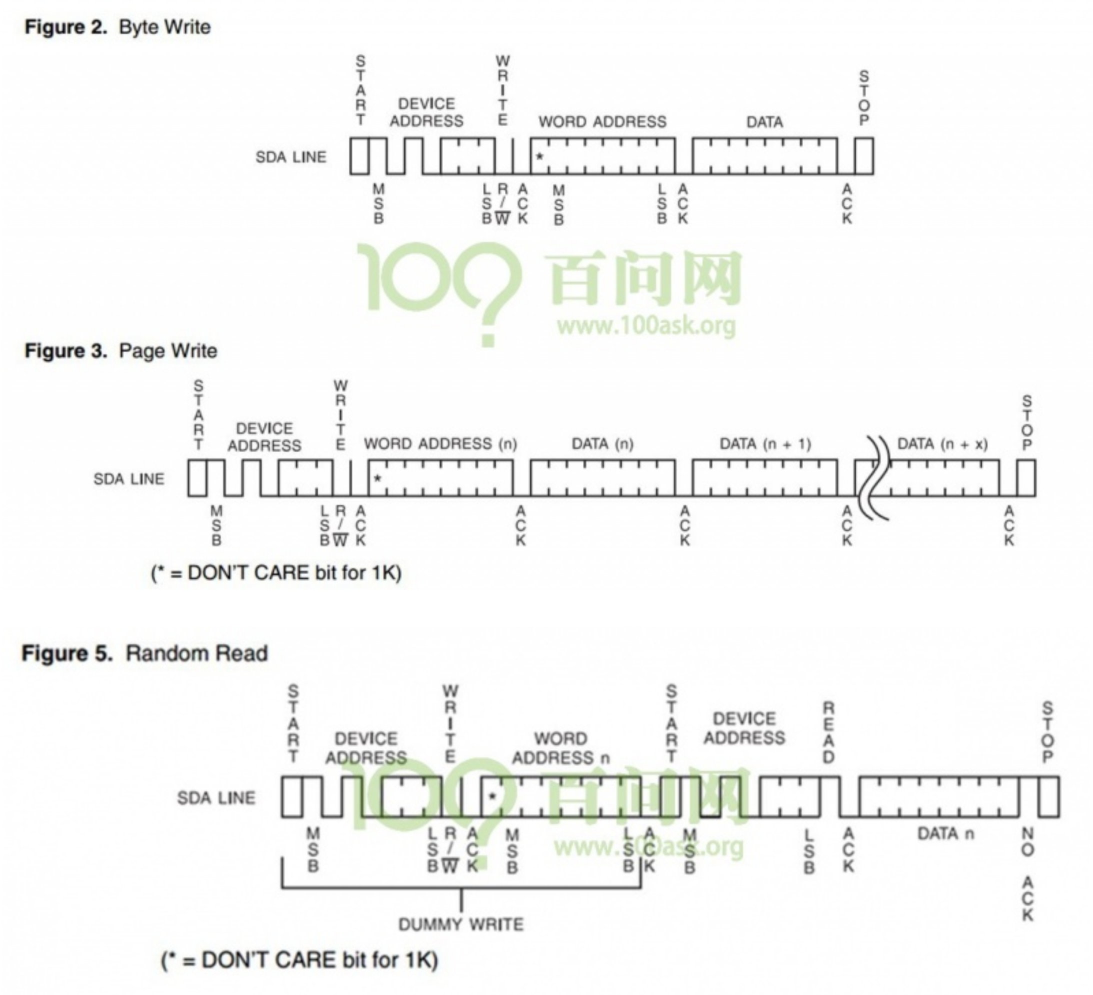
'



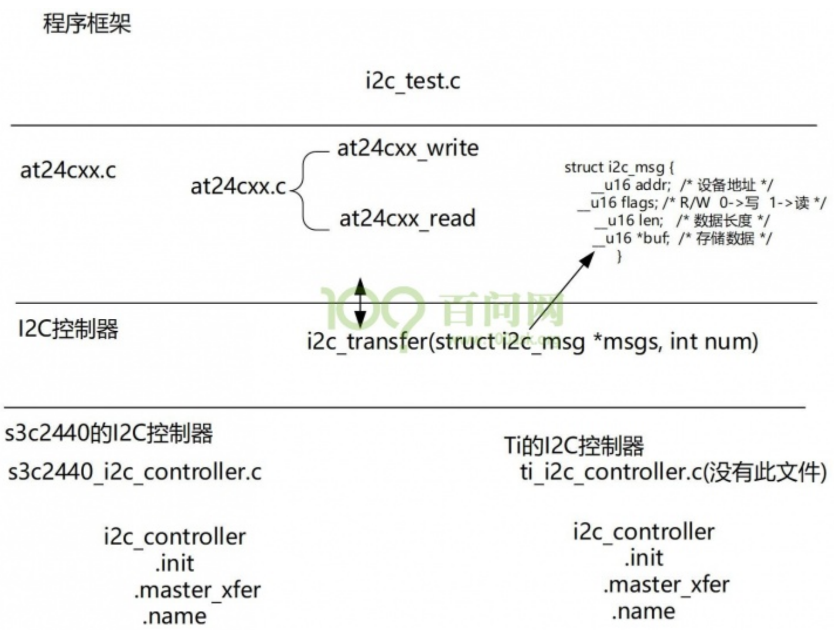
IIC 设备的功能

很显然，IIC 控制器提供了传输数据的能力，至于数据有什么含义，IIC 控制器并不知道，数据的含义有外接的 IIC 芯片决定，我们需要阅读芯片手册，才知道 IIC 控制器应该发出怎样的数据，

AT24cxx 的操作方法



显然我们的程序应该分为两层（IIC 设备层，IIC 控制器层），框架如下图



所示：

我们提供一个统一的接口 i2c_transfer，不关使用哪个芯片，他最终都会调用 i2c_transfer，来选择某一款 I2C 控制器，把数据发送出去，或者从 I2c 设备

读到数据，对于每一次传输的数据都可以用一个 i2c_msg 结构体来表示。但是，读某个地址的数据时，就要用两个 i2c_msg 结构体来描述它，因为一个 i2c_msg 结构体只能描述一个传输方向(读/写)，我们读取 ac24ccxx 某个地址上的数据时，要先写出要读取的地址，然后来读取设备地址上的数据。

我们想设计出以一个结构体比较容易扩展的框架，对于 I2C 控制器我们要抽象出一个结构体 i2c_controller，我们构造这个结构体之后，把这个这个结构体，告诉上层(I2C 控制器那一层)，上层有个管理者 i2c_contreller.c 文件。我们在 s3c2440_i2c_controller.c 这个文件中我们构造出一个 i2c_controller 结构体，把它放入上层文件中的数组里，以后就根据结构体的名字，把这个结构体取出来使用。假设我们有一个 TI 的开发板，在 ti_i2c_controller.c 文件中，也要构造出一个 i2c_controller 结构体，同样们也会把这个结构体放入上层的结构体数组(i2c_contreller.c 文件中)中，以后根据名字先出来使用。

对于设备层中的 at24cxx 芯片我们写出 at24cxx.c 文件在这个文件实现读写函数： 1，at24cxx_write 函数 2，at24cxx_read。函数读写函数都会调用 i2c_transfer 发起 IIC 传输，所以我们写程序的时候主要的暂时会涉及到三个文件：at24cxx.c， s3c2440_i2c_controller.c，i2c_contreller.c。在最上层会写出一个 i2c_test.c 文件，它会提供菜单供我们选择来测试。

下面我们写一个程序框架，涉及到的文件有：i2c_test.c at24cxx.c i2c_controller.c s3c2440_i2c_controller.c。**i2c_test.c 文件** 该文件的内容如下：

```
void i2c_test(void)
{
    /* 初始化: 选择I2C 控制器 */


    /* 提供菜单供测试 */
}
```

这个菜单最终会调用到 at24cxx.c 里面的函数。

at24cxx.c 文件 在里面会使用标准的接口 i2c_transfer 来启动 I2C 传输。该文件的内容如下：

```
int at24cxx_write(unsigned int addr, unsigned char *data, int len)
{
    /* 构造i2c_msg */


    /* 调用i2c_transfer */
}


int at24cxx_read(unsigned int addr, unsigned char *data, int len)
{
```

```
    /* 构造i2c_msg */

    /* 调用i2c_transfer */
}
```

i2c_controller.c 文件　该文件的内容如下：

```
/* 有一个i2c_controller 数组用来存放各种不同芯片的操作结构体
*/
void register_i2c_controller()
{
}

/* 根据名字来选择某款I2C 控制器 */
void select_i2c_controller(char *name)
{
}

/* 实现 i2c_transfer 接口函数 */

int i2c_transfer(i2c_msg msgs, int num)
{

}
```

select_i2c_controller 函数根据名字来选择某款 I2C 控制器后，以后就会使用被选择的 I2C 控制器来启动传输。有数组一定有注册函数 register_i2c_controller 会把下面实现的 I2C 控制器结构体 i2c_controller 放到 i2c_controller 数组里面。

s3c2440_i2c_controller.c 文件　对于具体的芯片，要实现自己的 i2c_controller。该文件的内容如下：

```
/* 实现i2c_controller
        .init
        .master_xfer
        .name
 */
```

第 004 节_I2C 控制器编程_框架

我们现在来讲 I2C 控制器怎么写，它是 I2C 程序中最核心的地方，我们要先构造几个结构体，这几个结构体放在 i2c_controller.h 里面。我们要发出 I2c 传输时，要构造出 i2c_msg，把构造出的 i2c_msg 扔给下面的 i2c_controller.c，i2c_controller.c 会选择某一个 i2c 控制器，使用里面的 master_xfer 来传输数据， 所以我们需要构造出一个 i2c_controller 结构体。 **i2c_controller.h 文件** 文件的内容如下所示：

```
#ifndef _I2C_CONTROLLER_H
#define _I2C_CONTROLLER_H

typedef struct i2c_msg {
    unsigned int addr;  /* 7bits */
    int flags;  /* 0 - write, 1 - read */
    int len;
    int cnt_transferred;
    unsigned char *buf;
}i2c_msg, *p_i2c_msg;

typedef struct i2c_controller {
    int (*int)(void);
    int (*master_xfer)(i2c_msg msgs, int num);
    char *name;
}i2c_controller, *p_i2c_controller;


#endif /* _I2C_CONTROLLER_H */
```

解析：我们构造这两个结构体，我们要把它放在 i2c_controller.c 把它用起来，

i2c_controller.c 文件 文件的内容如下所示：

```
include "i2c_controller.h"

#define I2C_CONTROLLER_NUM 10


/* 有一个i2c_controller 数组用来存放各种不同芯片的操作结构体
*/
static p_i2c_controller
p_i2c_controllers[I2C_CONTROLLER_NUM];
static p_i2c_controller p_i2c_con_selected;


void register_i2c_controller(p_i2c_controller *p)
{
```

```
    int i;
    for (i = 0; i < I2C_CONTROLLER_NUM; i++)
    {
        if (!p_i2c_controllers[i])
        {
            p_i2c_controllers[i] = p;
            return;
        }
    }
}
```

解析：register_i2c_controller 函数用于把参数中的结构体指针，注册到 p_i2c_controllers 指针数组中。

```
/* 根据名字来选择某款 I2C 控制器 */
int select_i2c_controller(char *name)
{
    int i;
    for (i = 0; i < I2C_CONTROLLER_NUM; i++)
    {
        if (p_i2c_controllers[i] && !strcmp(name,
p_i2c_controllers[i]->name))
        {
            p_i2c_con_selected =
p_i2c_controllers[i];
            return 0;
        }
    }
    return -1;
}
```

解析：select_i2c_controller 函数根据参数中的名字(name) 从 p_i2c_controllers 指针数组中取出对应的结构体指针复制给 p_i2c_con_selected 结构体指针(静态全局变量)。

```
/* 实现 i2c_transfer 接口函数 */

int i2c_transfer(i2c_msg msgs, int num)
{
    return p_i2c_con_selected->master_xfer(msgs, num);
}
```

解析：i2c_transfer 接口函数，调用选择的 p_i2c_con_selected 成员中 master_xfer 函数。

```
void i2c_init(void)
{
    /* 注册下面的 I2C 控制器 */
    s3c2440_i2c_con_add();

    /* 选择某款 I2C 控制器 */

    /* 调用它的 init 函数 */
}
```

解析：s3c2440_i2c_con_add()函数，把定义的 s3c2440_i2c_con 结构体注册到 p_i2c_controllers 数组中。

s3c2440_i2c_controller.c 文件 中断服务函数，当发成中断是，就会调用中断服务函数，代码如下

```
void i2c_interrupt_func(int irq)
{
    /* 每传输完一个数据将产生一个中断 */

    /* 对于每次传输，第 1 个中断是"已经发出了设备地址" */

}
```

s3c2440_i2c_con_init 函数，用来初始化 I2C，控制器代码如下：

```
void s3c2440_i2c_con_init(void)
{
    /* 设置时钟 */
    /* [7] : IIC-bus acknowledge enable bit, 1-enable
in rx mode
     * [6] : 时钟源, 0: IICCLK = fPCLK /16; 1: IICCLK =
fPCLK /512
```

```
        * [5] : 1-enable interrupt
        * [4] : 读出为1时表示中断发生了, 写入0来清除并恢复I2C
操作
        * [3:0] : Tx clock = IICCLK/(IICCON[3:0]+1).
        * Tx Clock = 100khz = 50Mhz/16/(IICCON[3:0]+1)
        */
       IICCON = (0<<6) | (1<<5) | (30<<0);

       /* 注册中断处理函数 */
       register_irq(27, i2c_interrupt_func);
   }
```

解析： 1)，IICCON = (0<<6) | (1<<5) | (30<<0); 设置 IICCON 控制寄存器。选择发送时钟，使能中断。 2），register_irq(27, i2c_interrupt_func)：注册中断处理函数，当发生 I2C 中断的时候就会调用 i2c_interrupt_func 中断处理函数。

初始化完成后，就可以调用 do_master_tx 写 I2C 从机了，这个函数仅仅启动 I2C 传输，然后等待，直到数据在中断服务程序中传输完毕后再返回。函数代码如下：

```
   void do_master_tx(p_i2c_msg msg)
   {
       msg->cnt_transferred = 0;

       /* 设置寄存器启动传输 */

       /* 1. 配置为 master tx mode */


       /* 2. 把从设备地址写入 IICDS */
       IICDS = msg->addr<<1;

       /* 3. IICSTAT = 0xf0 , 数据即被发送出去, 将导致中断产生
*/
       IICSTAT = 0xf0;


       /* 后续的传输由中断驱动 */
```

```c
        /* 循环等待中断处理完毕 */
        while (msg->cnt_transferred != msg->len);
    }
```

解析： 1），IICDS = msg->addr<<1: 把从机地址（高 7 位，所以需要向右移一位）写入到 IICDS 寄存器中。 2），IICSTAT = 0xf0:设置 IICSTAT 寄存器，将 s3c2440 设为主机发送器，并发出 S 信号后，紧接着就发出从机地址。后续的传输工作将在中断服务程序中完成。

do_master_rx 函数的实现和 do_master_tx 函数类似，代码如下：

```c
    void do_master_rx(p_i2c_msg msg)
    {
        msg->cnt_transferred = 0;

        /* 设置寄存器启动传输 */
        /* 1. 配置为 Master Rx mode */

        /* 2. 把从设备地址写入 IICDS */
        IICDS = (msg->addr<<1)|(1<<0);

        /* 3. IICSTAT = 0xb0 , 从设备地址即被发送出去, 将导致中
断产生 */
        IICSTAT = 0xb0;


        /* 后续的传输由中断驱动 */


        /* 循环等待中断处理完毕 */
        while (msg->cnt_transferred != msg->len);
    }
```

解析： 1），IICDS = (msg->addr<<1)|(1<<0)：把从设备地址写入 IICDS，前 7 位是从机地址，第 8 位表示传输方向(0 表示写操作，1 表示读操作)。

s3c2440 传输函数，根据标志位 flags，来指明是读/写(1：读 0：写)。代码如下：

```c
int s3c2440_master_xfer(p_i2c_msg msgs, int num)
{
	int i;
	for (i = 0; i < num; i++)
	{
			if (msgs[i]->flags == 0)/* write */
				do_master_tx(msgs[i]);
			else
				do_master_rx(msgs[i]);
	}
}
```

我们定义一个 i2c_controller 结构体 s3c2440_i2c_con。下面的代码对他进行初始化。

```c
static i2c_controller s3c2440_i2c_con = {
	.name = "s3c2440",
	.init = s3c2440_i2c_con_init,
	.master_xfer = s3c2440_master_xfer,
};
```

s3c2440_i2c_con_add 函数把上面定义的 s3c2440_i2c_con 结构体注册到上层的 i2c_controller 数组中。

```c
void s3c2440_i2c_con_add(void)
{
	register_i2c_controller(&s3c2440_i2c_con);
}
```

第 005 节_I2C 控制器编程_中断

中断控制器是 IIC 程序中的核心中的核心。Start 信号之后，发出设备地址，在第 9 个时钟就会产生一个中断，我们根据 i2c 的流程图来编写中断程序。每传输完一个数据将产生一个中断，I2C 操作的主体在中断服务程序，它可以分为两部分：写操作，读操作。先分析写操作，代码如下：

```c
void i2c_interrupt_func(int irq)
{
	int index;
	unsigned int iicstat = IICSTAT;

	p_cur_msg->cnt_transferred++;
```

```
/* 每传输完一个数据将产生一个中断 */

/* 对于每次传输, 第1个中断是"已经发出了设备地址" */

if (p_cur_msg->flags == 0)      /* write */
{
        /* 对于第1个中断, 它是发送出设备地址后产生的
         * 需要判断是否有ACK
         * 有ACK : 设备存在
         * 无ACK : 无设备, 出错, 直接结束传输
         */
        if (p_cur_msg->cnt_transferred == 0)  /* 第1次中断 */
        {
                if (iicstat & (1<<0))
                { /* no ack */
                        /* 停止传输 */
                        IICSTAT = 0xd0;
                        IICCON &= ~(1<<4);
                        p_cur_msg->err = -1;
                        delay(1000);
                        return;
                }
        }
```

1），p_cur_msg->cnt_transferred 初始值为-1(我们后面设置)。

2），p_cur_msg->cnt_transferred == 0 表示是第一次传输数据产生的中断，即发送从设备地址产生的中断。

3），iicstat & (1<<0)表示主机没有接受到 ACK 信号(即发出的设备地址不存在)，需要停止传输。

4），IICSTAT = 0xd0 置 IICSTAT 寄存器的[5]写为 0，以便发出 P 信号，但是由于这时 IICCON[4]仍为 1，P 信号没有实际发出，当执行 IICCON &= ~(1<<4);清除 IICCON[4]后，P 信号才真正发出。

5），等待一段时间，确保 P 信号已经发送完毕。

```
            if (p_cur_msg->cnt_transferred <
p_cur_msg->len)
            {
                /* 对于其他中断, 要继续发送下一个数据
                 */
                IICDS =
p_cur_msg->buf[p_cur_msg->cnt_transferred];
                IICCON &= ~(1<<4);
            }
            else
            {
                /* 停止传输 */
                IICSTAT = 0xd0;
                IICCON &= ~(1<<4);
                p_cur_msg->err = -1;
                delay(1000);
            }
        }
```

1），假如 if (p_cur_msg->cnt_transferred < p_cur_msg->len)条件成立，表示数据还没有发送完毕，需要继续发送数据。

2），执行 IICDS = p_cur_msg->buf[p_cur_msg->cnt_transferred 把要发送的数据写入到 IICDS 寄存器中，经过执行 IICCON &= ~(1<<4);清除中断标志后后，紧接着就自动把数据发送出去了，这将触发下一个中断。

3)，如果条件不成立表示数据传输完毕，发出 P 信号，停止数据的传输。

写操作：I2C 读操作的处理与写操作类似，我们就不进行分析了，代码如下：

```
    else /* read */
        {
            /* 对于第1 个中断, 它是发送出设备地址后产生的
             * 需要判断是否有ACK
             * 有ACK : 设备存在, 恢复I2C 传输, 这样在下一个中断
才可以得到第1 个数据
             * 无ACK : 无设备, 出错, 直接结束传输
             */
```

```
            if (p_cur_msg->cnt_transferred == 0)  /* 第1
次中断 */
            {
                  if (iicstat & (1<<0))
                  { /* no ack */
                        /* 停止传输 */
                        IICSTAT = 0x90;
                        IICCON &= ~(1<<4);
                        p_cur_msg->err = -1;
                        delay(1000);
                        return;
                  }
                  else  /* ack */
                  {
                        /* 恢复I2C 传输 */
                        IICCON &= ~(1<<4);
                        return;
                  }
            }

            /* 非第1 个中断，表示得到了一个新数据
             * 从IICDS 读出、保存
             */
            if (p_cur_msg->cnt_transferred <
p_cur_msg->len)
            {
                  index = p_cur_msg->cnt_transferred - 1;
                  p_cur_msg->buf[index] = IICDS;
                  /* 恢复I2C 传输 */
                  IICCON &= ~(1<<4);
            }
            else
            {
                  /* 发出停止信号 */
                  IICSTAT = 0x90;
                  IICCON &= ~(1<<4);
```

```
                delay(1000);
            }
        }
    }
```

我们还要在 s3c2440_i2c_con_init 函数中设置 IICCON 寄存器，设置 ACK 应答使能。

```
    IICCON = (1<<7) | (0<<6) | (1<<5) | (30<<0);
```

第 006 节_EEPROM 编程和测试代码

从设备程序，只涉及到两个函数分别是：从设备的写函数，从设备的读函数。下面下分析从设备的写函数，代码如下：

```
#define AT24CXX_ADDR 0x50

int at24cxx_write(unsigned int addr, unsigned char *data, int len)
{
    i2c_msg msg;
    int i;
    int err;
    unsigned char buf[2];


    for (i = 0; i < len; i++)
    {
        buf[0] = addr++;
        buf[1] = data[i];

        /* 构造i2c_msg */

        msg.addr  = AT24CXX_ADDR;
        msg.lags = 0; /* write */
        msg.len   = 2;
        msg.buf   = buf;
        msg.err   = 0;
        msg.cnt_transferred = -1;

        /* 调用i2c_transfer */

        err = i2c_transfer(&msg, 1);
        if (err)
            return err;
```

```
    }

    return 0;
}
```

1），#define AT24CXX_ADDR 0x50 宏定义设备地址。

2），我们每次只写一个字节，所以我们需要构造出 len 个 msg。

3），调用 i2c 接口函数，传输构造 i2C_msg 结构体，我们传输指针只需要传输四个字节，我们需要把以前的参数都改成传输指针的格式。

从设备读函数和写函数类似，读函数需要构造两个 i2c_msg(每个 i2c_msg 只能表示一个传输方向），因为在读操作之前，需要把要读的地址告诉从设备。代码如下：

```
int at24cxx_read(unsigned int addr, unsigned char *data, int len)
{
    i2c_msg msg[2];
    int err;

    /* 构造i2c_msg */
    msg[0].addr  = AT24CXX_ADDR;
    msg[0].lags  = 0; /* write */
    msg[0].len   = 1;
    msg[0].buf   = &addr;

    msg[0].err   = 0;
    msg[0].cnt_transferred = -1;

    msg[1].addr  = AT24CXX_ADDR;
    msg[1].lags  = 1; /* read */
    msg[1].len   = len;
    msg[1].buf   = data;
    msg[1].err   = 0;
    msg[1].cnt_transferred = -1;

    /* 调用i2c_transfer */
    err = i2c_transfer(&msg, 2);
    if (err)
        return err;
    return 0;
}
```

I2c_test 测试程序如下所示：

```c
void i2c_test(void)
{
    char c;

    /* 初始化 */
    i2c_init();

    while (1)
    {
        /* 打印菜单, 供我们选择测试内容 */
        printf("[w] Write at24cxx\n\r");
        printf("[r] Read at24cxx\n\r");
        printf("[q] quit\n\r");
        printf("Enter selection: ");

        c = getchar();
        printf("%c\n\r", c);

        /* 测试内容:
         * 3. 编写某个地址
         * 4. 读某个地址
         */
        switch (c)
        {
            case 'q':
            case 'Q':
                return;
                break;

            case 'w':
            case 'W':
                do_write_at24cxx();
                break;

            case 'r':
            case 'R':
                do_read_at24cxx();
                break;
            default:
```

```
                    break;
            }
        }
    }
```

1），调用 i2c_controller.c 里面的 i2c_init()初始化函数，在这个函数中需要添加一些功能，i2c_init()代码如下所示：

```
    void i2c_init(void)
    {
        /* 注册下面的I2C 控制器 */
        s3c2440_i2c_con_add();

        /* 选择某款I2C 控制器 */
        select_i2c_controller("s3c2440");

        /* 调用它的init 函数 */
        p_i2c_con_selected->init();
    }
```

- select_i2c_controller("s3c2440")用于选择 s3c2440 的 i2c 控制器。
- p_i2c_con_selected->init()调用 s3c2440 的 i2c 控制器结构体中 init 初始化函数，初始化 s3c2440 的 i2c 控制器。

2），执行 do_write_at24cxx()函数用于往 at24cxx 设备中写入数据，do_write_at24cxx()函数的代码如下所示：

```
    {
        unsigned int addr;
        unsigned char str[100];
        int err;

        /* 获得地址 */
        printf("Enter the address of sector to write: ");
        addr = get_uint();

        if (addr > 256)
        {
            printf("address > 256, error!\n\r");
            return;
        }
```

```
    printf("Enter the string to write: ");
    gets(str);

    printf("writing ...\n\r");
    err = at24cxx_write(addr, str, strlen(str)+1);
    printf("at24cxx_write ret = %d\n\r", err);
}
```

- addr = get_uint()用于把输入的地址赋值给 addr。
- gets(str)用于把输入的字符串存在 str 字符数组中。
- at24cxx_write(addr, str, strlen(str)+1)调用 at24cxx_write 函数把输入的数据 str,，放在输入的地址 addr 中。

3），执行 do_read_at24cxx()函数从 at24cxx 中读取数据，do_read_at24cxx()函数的代码如下所示：

```
void do_read_at24cxx(void)
{
    unsigned int addr;
    int i, j;
    unsigned char c;
    unsigned char data[100];
    unsigned char str[16];
    int len;
    int err;
    int cnt = 0;


    /* 获得地址 */
    printf("Enter the address to read: ");
    addr = get_uint();

    if (addr > 256)
    {
          printf("address > 256, error!\n\r");
          return;
    }


    /* 获得长度 */
    printf("Enter the length to read: ");
    len = get_int();

    err = at24cxx_read(addr, data, len);
    printf("at24cxx_read ret = %d\n\r", err);
```

```c
        printf("Data : \n\r");
        /* 长度固定为64 */
        for (i = 0; i < 4; i++)
        {
            /* 每行打印16 个数据 */
            for (j = 0; j < 16; j++)
            {
                /* 先打印数值 */
                c = data[cnt++];
                str[j] = c;
                printf("%02x ", c);
            }

            printf("   ; ");

            for (j = 0; j < 16; j++)
            {
                /* 后打印字符 */
                if (str[j] < 0x20 || str[j] > 0x7e)  /*
不可视字符 */
                    putchar('.');
                else
                    putchar(str[j]);
            }
            printf("\n\r");
        }
    }
```

- 调用 at24cxx_read(addr, data, len)函数，从 addr 地址中读取 len 长度的字节数据，放在 data 字符数组中，后面的代码就是把读取得到的数据，打印出来。

第 007 节_测试

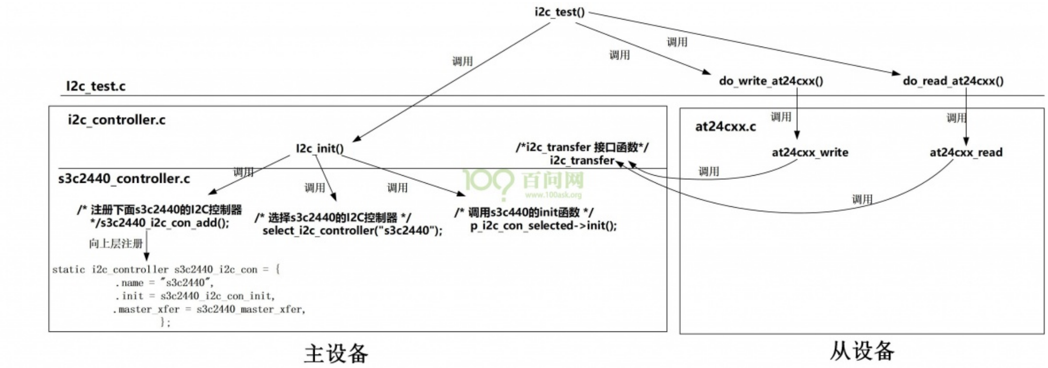
在测试中，出现问题和解决办法：

a 中断没产生 ：未配置 GPIO 用于 IIC 功能 解决方法：配置引脚用于 I2C

b. 只产生了一次中断，并且出错 ：tx err，no ack 解决方法：启动传输之前 IICSTAT=(1<<4)

c. 第 1 次读 OK，再次写卡死，复位再写仍卡死，重新上电再写 OK：解决方法：读最后一个数据时，不要回应 ACK 给 AT24CXX

程序框架如下图所示：



第 20 课 SPI

第 001 节_SPI 协议介绍

配套视频:ARM 裸机 1 期加强版->第 20 课_SPI->第 001 节_SPI 协议介绍_P.mp4

配套代码:023_spi_020->01th_spi_i2c_adc_jz2440_oled_020_002

配套笔记:

实验环境:百问网 Ubuntu16.04、Window7/Window10

适用单板:JZ2440(原理适用所有 Soc)

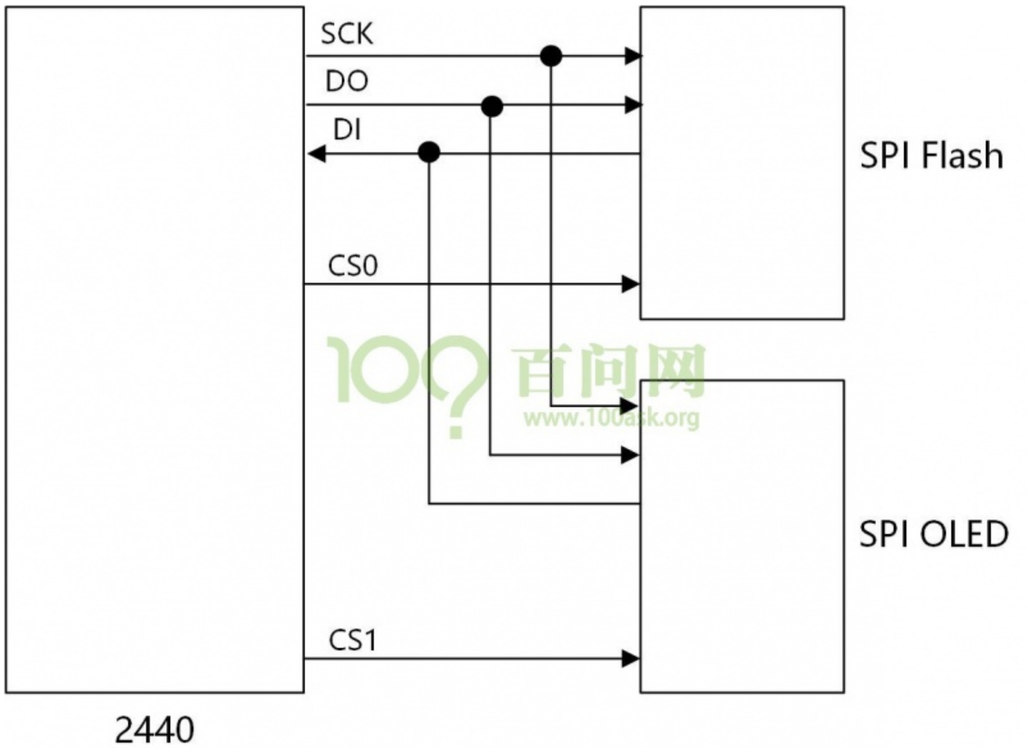
市面上的开发板很少接有 SPI 设备，但是 SPI 协议在工作中经常用到。我们开发了 SPI 模块，上面有 SPI Flash 和 SPI OLED。OLED 就是一块显示器。

我们裸板程序会涉及两部分：

用 GPIO 模拟 SPI

用 S3C2440 的 SPI 控制器

我们先介绍下 SPI 协议，硬件框架如



下：

SCK：提供时钟

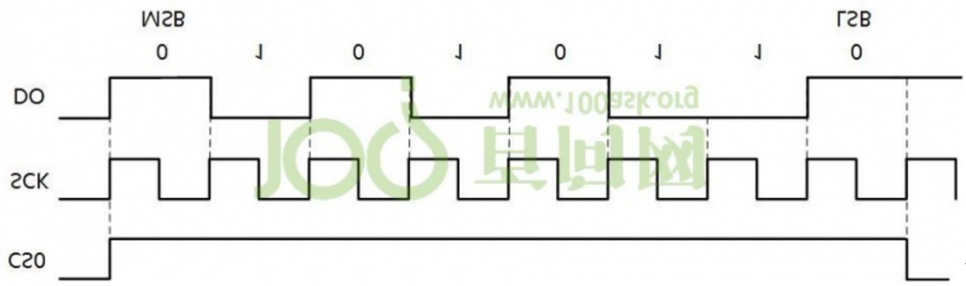
DO:作为数据输出

DI:作为数据输入

CS0/CS1:作为片选

同一时刻只能有一个 SPI 设备处于工作状态。

假设现在 2440 传输一个 0x56 数据给 SPI Flash，时序如下：



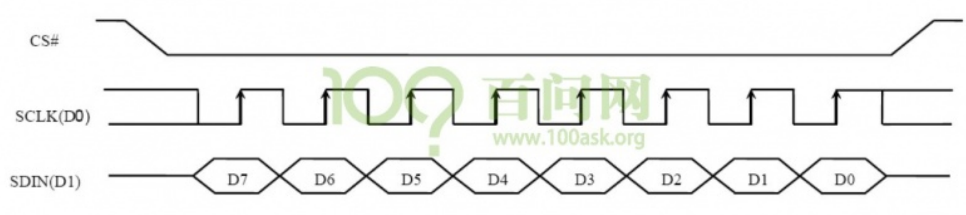
首先 CS0 先拉低选中 SPI Flash，0x56 的二进制就是 0b0101 0110，因此在每个 SCK 时钟周期，DO 输出对应的电平。 SPI Flash 会在每个时钟周期的上升沿读取 D0 上的电平。

在 SPI 协议中，有两个值来确定 SPI 的模式。 CPOL:表示 SPICLK 的初始电平，0 为电平，1 为高电平 CPHA:表示相位，即第一个还是第二个时钟沿采样数据，0 为第一个时钟沿，1 为第二个时钟沿

CPOL	CPHA	模式	含义
0	0	0	初始电平为低电平，在第一个时钟沿采样数据
0	1	1	初始电平为低电平，在第二个时钟沿采样数据
1	0	2	初始电平为高电平，在第一个时钟沿采样数据
1	1	3	初始电平为高电平，在第二个时钟沿采样数据

我们常用的是模式 0 和模式 3，因为它们都是在上升沿采样数据，不用去在乎时钟的初始电平是什么，只要在上升沿采集数据就行。

极性选什么？格式选什么？通常去参考外接的模块的芯片手册。比如对于 OLED，查看它的芯片手册时序部分：

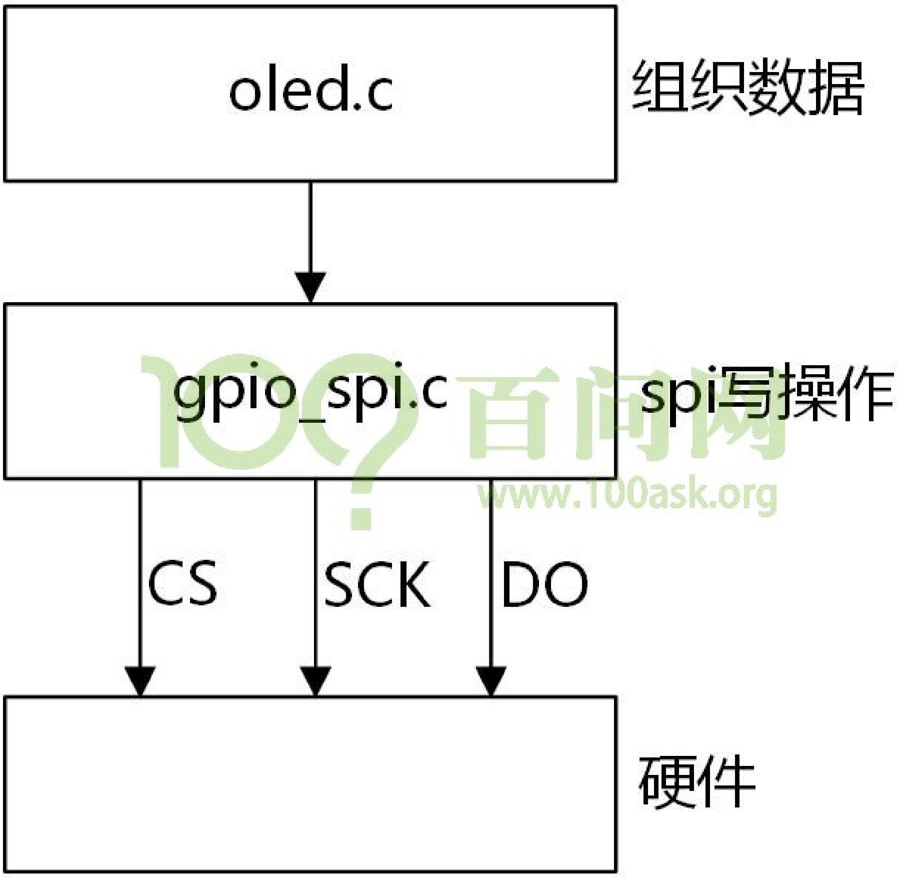


SCLK 的初始电平我们并不需要关心，只要保证在上升沿采样数据就行。

第 002 节_使用 GPIO 实现 SPI 协议操作

OLED

现在开始写代码，使用 GPIO 实现 SPI 协议操作。 我们现在想要操作 OLED，通过三条线(SCK、DO、CS)与 OLED 相连，这里没有 DI 是因为 2440 只会向 OLED 传数据而不用接收数据。 我们要用 GPIO 来实现 SOC 向 OLED 写数据，这一层用 gpio_spi.c 来实现，负责发送数据。 对于 OLED，有专门的指令和数据格式，要传输的数据内容，在 oled.c 这一层来实现，负责组织数据。 因此，我们需要实现以上两个文件。

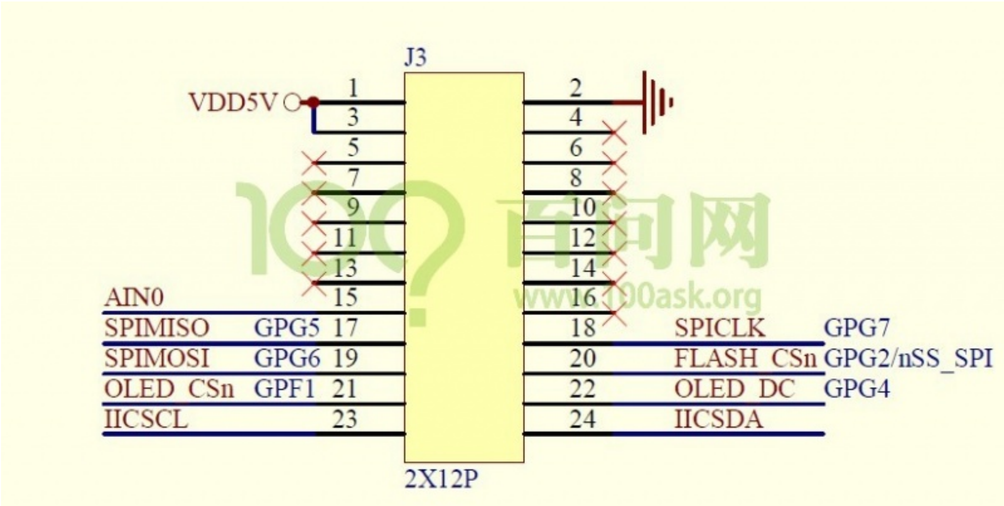


需要实现的函数：先 SPI 初始化 SPIInt()，再初始化 OLEDOLEDInit()，最后再显示 OLEDPrint()。

新建一个 gpio_spi.c 文件，实现 SPI 初始化 SPIInt()

```
void SPIInit(void)
{
    /* 初始化引脚 */
    SPI_GPIO_Init();
}
```

再具体实现 SPI_GPIO_Init()。这里使用 GPIO 实现 SPI 协议，电路图如下：



GPF1 作为 OLED 片选引脚，设置为输出；
GPG2 作为 FLASH 片选引脚，设置为输出；
GPG4 作为 OLED 的数据(Data)/命令(Command)选择引脚，设置为输出；
GPG5 作为 SPI 的 MISO，设置为输入；
GPG6 作为 SPI 的 MOSI，设置为输出；
GPG7 作为 SPI 的时钟 CLK，设置为输出；

```
/* 用GPIO模拟SPI */
static void SPI_GPIO_Init(void)
{
    /* GPF1 OLED_CSn output */
    GPFCON &= ~(3<<(1*2));
    GPFCON |= (1<<(1*2));
    GPFDAT |= (1<<1);

    /* GPG2 FLASH_CSn output
     * GPG4 OLED_DC   output
     * GPG5 SPIMISO   input
     * GPG6 SPIMOSI   output
     * GPG7 SPICLK    output
     */
    GPGCON &= ~((3<<(2*2)) | (3<<(4*2)) | (3<<(5*2)) |
(3<<(6*2)) | (3<<(7*2)));
    GPGCON |= ((1<<(2*2)) | (1<<(4*2)) | (1<<(6*2)) |
(1<<(7*2)));
    GPGDAT |= (1<<2);
}
```

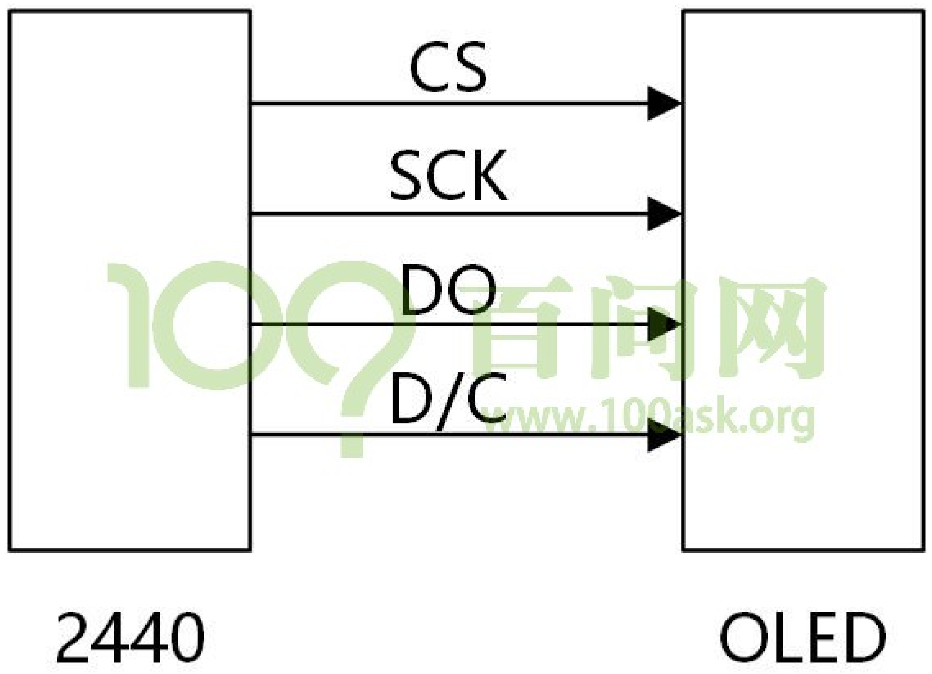
再新建一个 oled.c 文件，以实现初始化 OLEDOLEDInit()

```c
void OLEDInit(void)
{
/* 向OLED 发命令以初始化 */
}
```

查阅 OLED 数据手册 SPEC UG-2864TMBEG01.pdf 可以得知其初始化流程和参考的初始化代码：

```c
void OLEDInit(void)
{
    /* 向OLED 发命令以初始化 */
    OLEDWriteCmd(0xAE); /*display off*/
    OLEDWriteCmd(0x00); /*set lower column address*/
    OLEDWriteCmd(0x10); /*set higher column address*/
    OLEDWriteCmd(0x40); /*set display start line*/
    OLEDWriteCmd(0xB0); /*set page address*/
    OLEDWriteCmd(0x81); /*contract control*/
    OLEDWriteCmd(0x66); /*128*/
    OLEDWriteCmd(0xA1); /*set segment remap*/
    OLEDWriteCmd(0xA6); /*normal / reverse*/
    OLEDWriteCmd(0xA8); /*multiplex ratio*/
    OLEDWriteCmd(0x3F); /*duty = 1/64*/
    OLEDWriteCmd(0xC8); /*Com scan direction*/
    OLEDWriteCmd(0xD3); /*set display offset*/
    OLEDWriteCmd(0x00);
    OLEDWriteCmd(0xD5); /*set osc division*/
    OLEDWriteCmd(0x80);
    OLEDWriteCmd(0xD9); /*set pre-charge period*/
    OLEDWriteCmd(0x1f);
    OLEDWriteCmd(0xDA); /*set COM pins*/
    OLEDWriteCmd(0x12);
    OLEDWriteCmd(0xdb); /*set vcomh*/
    OLEDWriteCmd(0x30);
    OLEDWriteCmd(0x8d); /*set charge pump enable*/
    OLEDWriteCmd(0x14);
}
```

因此我们还要先实现 OLEDWriteCmd()函数，对于 OLED，除了 SPI 的片选、时钟、数据引脚，还有一个数据/命令切换引脚。



这里的 D/C 即数据(Data)/命令(Command)选择引脚，它为高电平时，OLED 即认为收到的是数据；它为低电平时，OLED 即认为收到的是命令。 对于 OLED，命令由开启/关闭显示、背光亮度等，具体有什么命令，可以查阅 OLED 的主控芯片手册 SSD1306-Revision 1.1 (Charge Pump).pdf，在 9 COMMAND TABLE 有相关命令的介绍。 因此，在编写 OLEDWriteCmd()时，需要先设置为命令模式：

```
static void OLEDWriteCmd(unsigned char cmd)
{
    OLED_Set_DC(0); /* command */
    OLED_Set_CS(0); /* select OLED */

    SPISendByte(cmd);

    OLED_Set_CS(1); /* de-select OLED */
    OLED_Set_DC(1); /*  */
}
```

即：先设置为命令模式，再片选 OLED，再传输命令，再恢复成原来的模式和取消片选。

片选函数和模式切换函数都比较简单，设置为对应的高低电平即可：

```
static void OLED_Set_DC(char val)
{
    if (val)
        GPGDAT |= (1<<4);
```

```c
    else
        GPGDAT &= ~(1<<4);
}

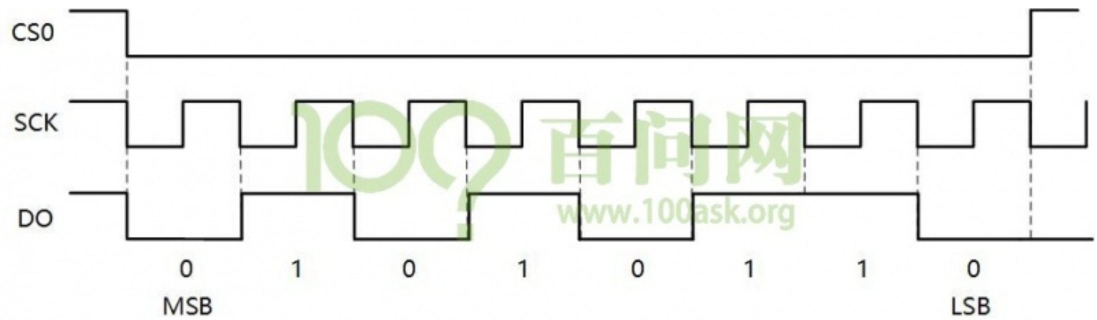
static void OLED_Set_CS(char val)
{
    if (val)
        GPFDAT |= (1<<1);
    else
        GPFDAT &= ~(1<<1);
}
```

还剩下 SPISendByte()函数，它属于 SPI 协议，放在 gpio_spi.c 里面：

```c
void SPISendByte(unsigned char val)
{
    int i;
    for (i = 0; i < 8; i++)
    {
        SPI_Set_CLK(0);
        SPI_Set_DO(val & 0x80);
        SPI_Set_CLK(1);
        val <<= 1;
    }

}
```

发送数据要满足 SPI 的时序要求，参考前面的介绍：



先设置 CLK 为低，然后数据引脚输出数据的最高位，然后 CLK 为高，在 CLK 这个上升沿中,OLED 就读取了一位数据。接着左移一位，将原来的第 7 位移动到了第 8 位，重复 8 次，传输完成。

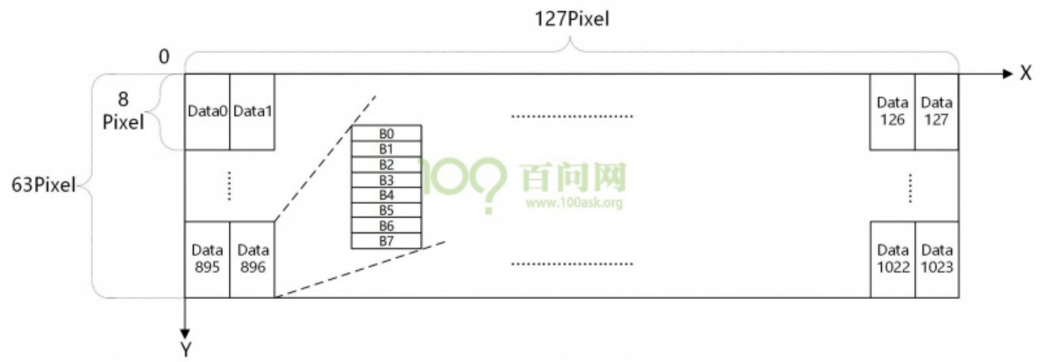
再完成 SPI_Set_CLK()和 SPI_Set_DO()：

```c
static void SPI_Set_CLK(char val)
{
    if (val)
        GPGDAT |= (1<<7);
```

```
    else
        GPGDAT &= ~(1<<7);
}

static void SPI_Set_DO(char val)
{
    if (val)
        GPGDAT |= (1<<6);
    else
        GPGDAT &= ~(1<<6);
}
```

至此，SPI 初始化和 OLED 初始化就基本完成了，接下来就是 OLED 显示部分。 先了解一下 OLED 显示的原理：



OLED 长有 128 个像素，宽有 64 个像素，每个像素用一位来表示，为 1 则亮，为 0 则灭。 每一个字节数据 Datax 控制每列 8 个像素，在显存里面存放 Data 数据。 之后所需的操作就是把数据写到显存里面去，如何写到显存可以拆分成两个问题： ①怎么发地址 ②怎么发数据

OLED 主控的手册里介绍了三种地址模式，我们常用的是页地址模式(Page addressing mode (A[1:0]=10xb))，它把显存的 64 行分为 8 页，每页对应 8 行；选中某页后，再选择某列，然后就可以往里面写数据了，每写一个数据，地址就会加 1，一直写到最右端的位置，他会自动跳到最左端。 通过命令来实现发送页地址和列地址，其中列地址分为两次发送，先发送低字节，再发送高字节。

假设每个字符数据大小为 8x16，假如第一个字符位置为(page,col)，相邻的右边就是(page,col+8)，写满一行跳至下一行的坐标就是(page+2,col)。

```
/* page: 0-7
 * col : 0-127
 * 字符: 8x16 象素
 */
void OLEDPrint(int page, int col, char *str)
{
    int i = 0;
    while (str[i])
```

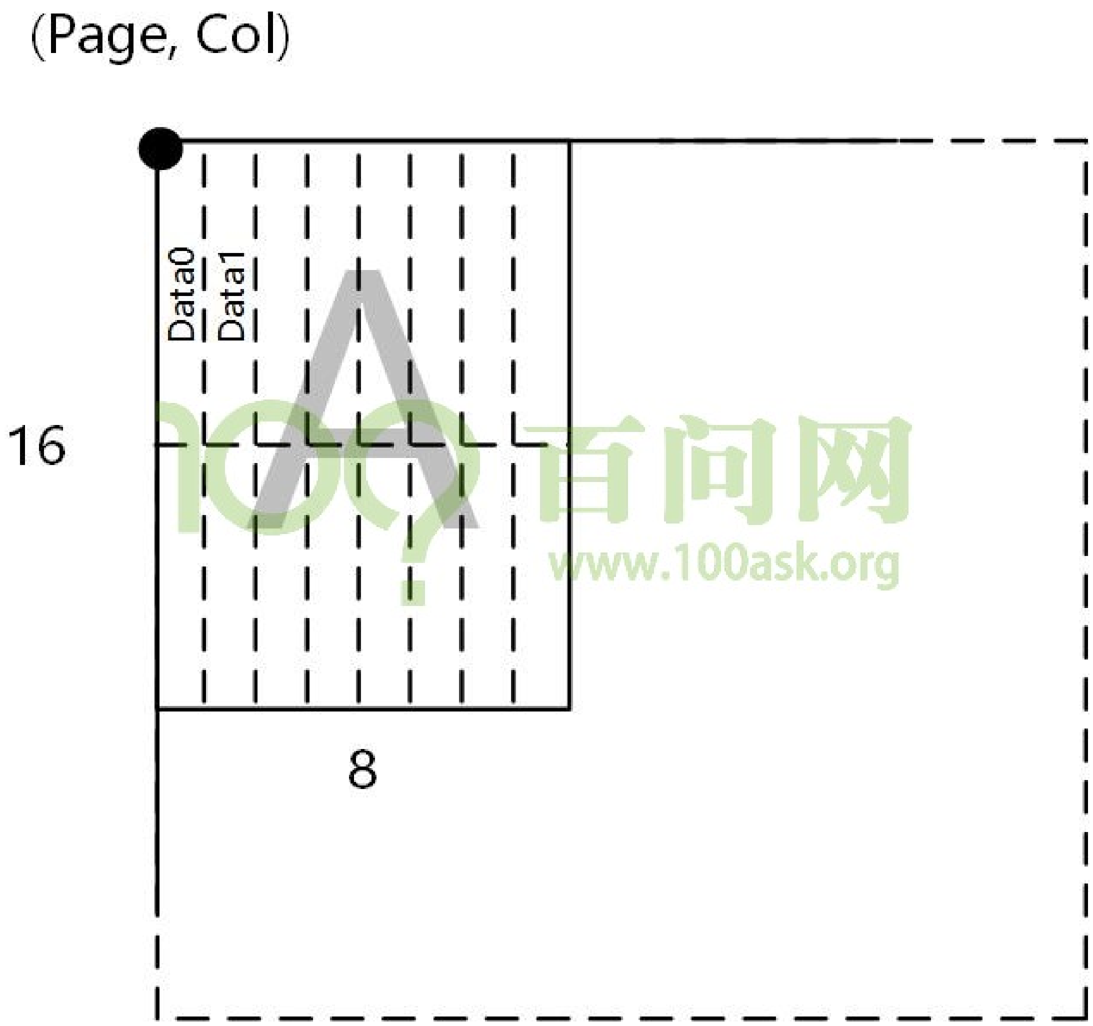
```
        {
            OLEDPutChar(page, col, str[i]);
            col += 8;
            if (col > 127)
            {
                col = 0;
                page += 2;
            }
            i++;
        }
    }
```

只要字符数组 str[i]有数据，就调用 OLEDPutChar(page, col, str[i])在指定位置显示第一个字符，然后位置向右移动一个字符的大小，如果遇到行尾，再进行换行，就这样依次显示完所有字符。

现在开始实现最重要的 OLEDPutChar()函数。把一个字符在 OLED 上显示出来需要以下几个步骤： a. 得到字模 b. 发给 OLED

字模我们可以从网上搜索相关资料获取到，将字模的数组 oled_asc2_8x16[95][16]放在 oledfont.c 里面，字符从空格开始，因此每次减去一个空格才是我们想要的字符。

如图所示一个字符，先以(page, col)为起点，显示 8 位数据，再换行，以(page+1, col)为起点显示 8 位数据。




```
/* page: 0-7
 * col : 0-127
 * 字符: 8x16 象素
 */
void OLEDPutChar(int page, int col, char c)
{
    int i = 0;
    /* 得到字模 */
    const unsigned char *dots = oled_asc2_8x16[c - ' '];

    /* 发给 OLED */
    OLEDSetPos(page, col);
    /* 发出 8 字节数据 */
```

```c
        for (i = 0; i < 8; i++)
            OLEDWriteDat(dots[i]);

        OLEDSetPos(page+1, col);
        /* 发出 8 字节数据 */
        for (i = 0; i < 8; i++)
            OLEDWriteDat(dots[i+8]);
    }
```

显示一个字符，就先获取字模数据，接着发出 8 字节数据，再换行发出 8 字节数。

再来实现 OLED 设置坐标位置函数，先设置 page：

D/C#	Hex	D7	D6	D5	D4	D3	D2	D1	D0	Command	Description
0	B0~B7	1	0	1	1	0	X_2	X_1	X_0	Set Page Start Address for Page Addressing Mode	Set GDDRAM Page Start Address (PAGE0~PAGE7) for Page Addressing Mode using X[2:0]. **Note** (1) This command is only for page addressing mode

3. Addressing Setting Command Table

D0~D2 表示 page 数据，D3-D7 是固定的值，因此每次写的命令内容为 0xB0+page；

再设置列：

D/C#	Hex	D7	D6	D5	D4	D3	D2	D1	D0	Command	Description
0	00~0F	0	0	0	0	X_3	X_2	X_1	X_0	Set Lower Column Start Address for Page Addressing Mode	Set the lower nibble of the column start address register for Page Addressing Mode using X[3:0] as data bits. The initial display line register is reset to 0000b after RESET. **Note** (1) This command is only for page addressing mode
0	10~1F	0	0	0	1	X_3	X_2	X_1	X_0	Set Higher Column Start Address for Page Addressing Mode	Set the higher nibble of the column start address register for Page Addressing Mode using X[3:0] as data bits. The initial display line register is reset to 0000b after RESET. **Note** (1) This command is only for page addressing mode

3. Addressing Setting Command Table

分两次发送，显示发送低字节 4 位，再发送高字节四位；

```c
    static void OLEDSetPos(int page, int col)
    {
        OLEDWriteCmd(0xB0 + page); /* page address */

        OLEDWriteCmd(col & 0xf);   /* Lower Column Start
Address */
        OLEDWriteCmd(0x10 + (col >> 4));   /* Lower Higher
Start Address */
    }
```

前面提到了 OLED 主控有三种地址模式，我们常用的是页地址模式(Page addressing mode (A[1:0]=10xb))，虽然这是默认的摸索，但还是设置一下比较好：

0	20	0	0	1	0	0	0	0	0	Set Memory	A[1:0] = 00b, Horizontal Addressing Mode
0	A[1:0]	*	*	*	*	*	*	A1	A0	Addressing Mode	A[1:0] = 01b, Vertical Addressing Mode A[1:0] = 10b, Page Addressing Mode (RESET) A[1:0] = 11b, Invalid

即先发送 0x20，再设置 A[1:0]=10：

```
static void OLEDSetPageAddrMode(void)
{
    OLEDWriteCmd(0x20);
    OLEDWriteCmd(0x02);
}
```

在显示中，一般都需一个清屏函数来清空当前可能显示的数据。清屏函数比较简单，往所有位置里面写 0 即可：

```
static void OLEDClear(void)
{
    int page, i;
    for (page = 0; page < 8; page ++)
    {
        OLEDSetPos(page, 0);
        for (i = 0; i < 128; i++)
            OLEDWriteDat(0);
    }
}
```

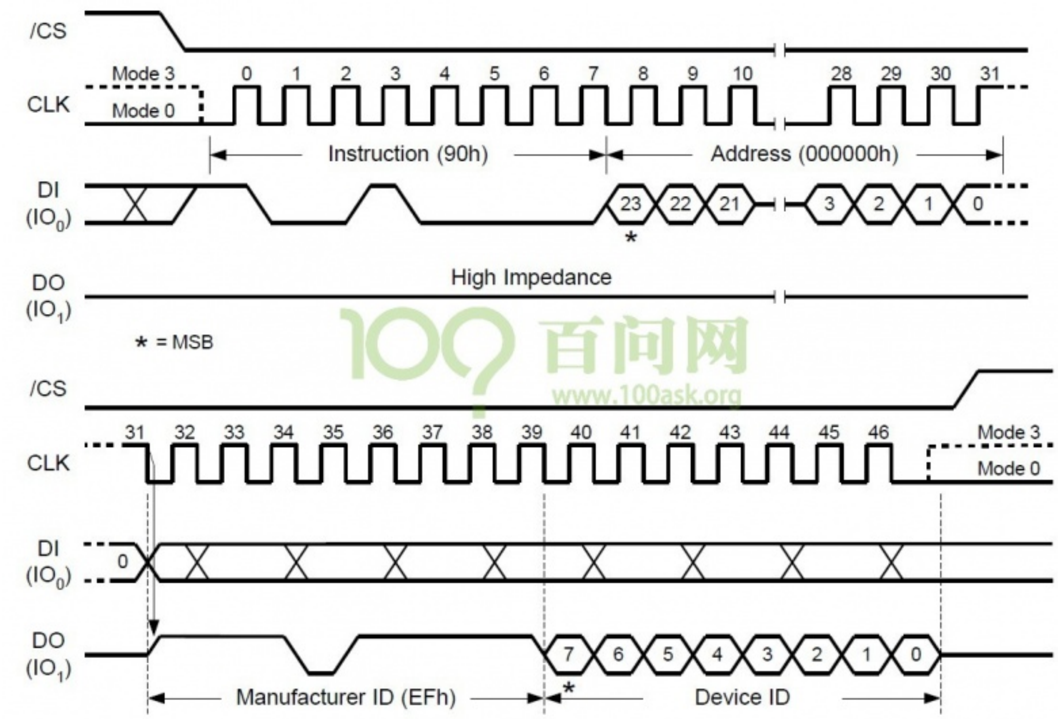
再把地址模式 OLEDSetPageAddrMode()和清屏函数 OLEDClear()放在 SPI_GPIO_Init()里，在 Makefile 加上 gpio_spi.o 和 oled.o。
最后在主函数里加上初始化和显示函数：

```
SPIInit();
OLEDInit();
OLEDPrint(0,0,"www.100ask.net, 100ask.taobao.com");
```

第 003 节_SPI_FLASH 编程_读 ID

这节讲解如何使用 SPI 操作 Flash，我们在上节课的代码上进行修改，添加一个文件 spi_flash.c 和其头文件 spi_flash.h 。 我们先做一个最简单的 spi 操作，读取 Flash 的 ID， SPIFlashID() 。 Flash 的 ID 有厂家 ID 和设备 ID，分别用 pMID 和 pDID 来保存。 根据 Flash 的芯片手册 W25Q16DV.pdf 可以

知道需要先发出一个指令 0x90，再发送 24 位的地址 0，再读取数据前 8 位是设备 ID，然后是 8 位设备 ID。进行操作前必须要片选 SPI Flash，片选完还是释放 SPI Flash:




```
void SPIFlashReadID(int *pMID, int *pDID)
{
    SPIFlash_Set_CS(0); /* 选中SPI FLASH */

    SPISendByte(0x90);

    SPIFlashSendAddr(0);

    *pMID = SPIRecvByte();
    *pDID = SPIRecvByte();

    SPIFlash_Set_CS(1);
}
```

把其中的发送 24 地址封装成了一个函数 SPIFlashSendAddr()　:

```
static void SPIFlashSendAddr(unsigned int addr)
{
    SPISendByte(addr >> 16);
    SPISendByte(addr >> 8);
    SPISendByte(addr & 0xff);
}
```

依次完成上面的子函数，先是 SPI 片选，上一节的原理图可以看到 SPI Flash 的片选是 GPG2:

```
static void SPIFlash_Set_CS(char val)
{
    if (val)
        GPGDAT |= (1<<2);
    else
        GPGDAT &= ~(1<<2);
}
```

SPISendByte() 和前面 OLED 的是一样的，就不用写了，因此就只剩下 SPIRecvByte() ，放在 gpio_spi.c 里面实现:

```
unsigned char SPIRecvByte(void)
{
    int i;
    unsigned char val = 0;
    for (i = 0; i < 8; i++)
    {
        val <<= 1;
        SPI_Set_CLK(0);
        if (SPI_Get_DI())
            val |= 1;
        SPI_Set_CLK(1);
    }
    return val;
}
```

在每个时钟周期读取 DI 引脚上的值，对于 SOC 就是 MISO 引脚:

```
static char SPI_Get_DI(void)
{
    if (GPGDAT & (1<<5))
        return 1;
    else
        return 0;
}
```

至此，读取 Flash 的 ID 基本实现，最后在主函数里调用打印，分别在串口和 OLED 上显示:

```
        SPIFlashReadID(&mid, &pid);
        printf("SPI Flash : MID = 0x%02x, PID = 
0x%02x\n\r", mid, pid);
```

```
    sprintf(str, "SPI : %02x, %02x", mid, pid);
    OLEDPrint(4,0,str);
```

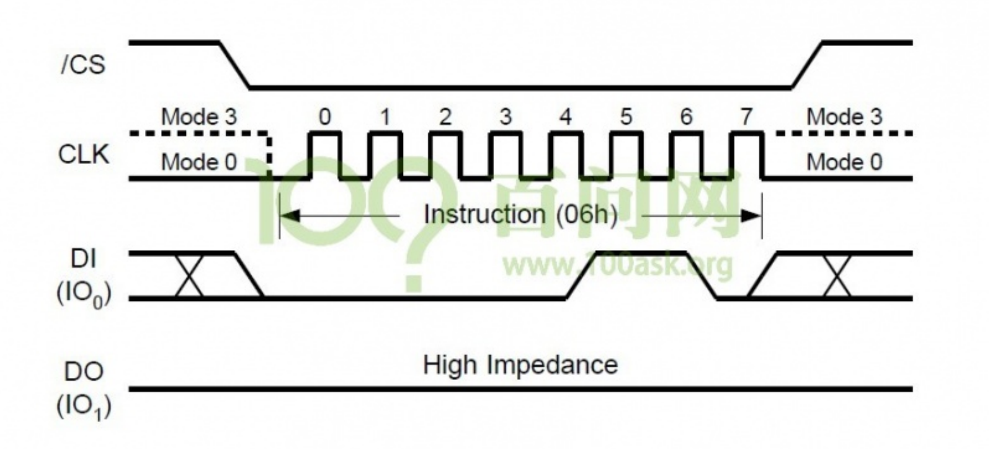
Makefile 记得加上新生成的 spi_flash.o 。

第 004 节_SPI_FLASH 编程_读写

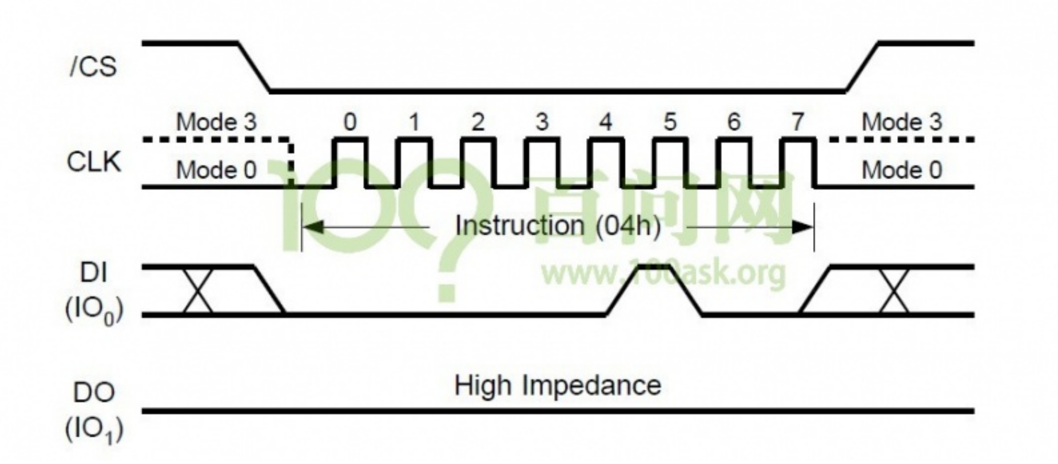
Flash 作为一个存储芯片，最重要的就是存储和读取存储的数据，这节我们就实现 Flash 里数据的读写。 对于 Flash，每次写操作需要的步骤如下：

1. 去保护（写使能、写状态寄存器）；
2. 擦除（写使能）
3. 编写入数据（写使能）

可以看出对于写操作，每次都要写使能，查阅芯片手册，可以看出写使能比较简单，只需要发送 0x06 命令即可：



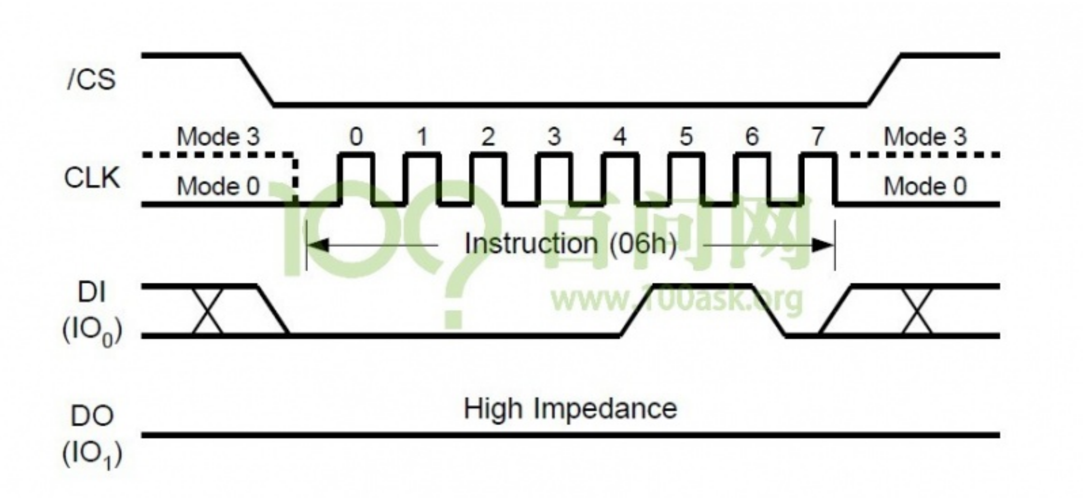
反之，写保护则是写入 0x04：




```
static void SPIFlashWriteEnable(int enable)
{
```

```
    if (enable)
    {
        SPIFlash_Set_CS(0);
        SPISendByte(0x06);
        SPIFlash_Set_CS(1);
    }
    else
    {
        SPIFlash_Set_CS(0);
        SPISendByte(0x04);
        SPIFlash_Set_CS(1);
    }
}
```

然后是读写状态寄存器，状态寄存器有两个，通过 0x05 读取状态寄存器 1，通过 0x35 读取状态寄存器 2：

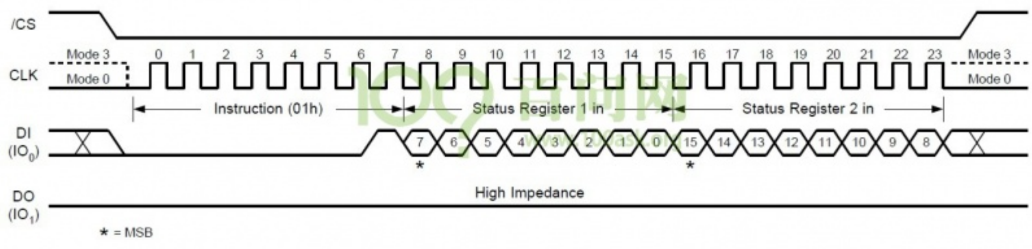



```
static unsigned char SPIFlashReadStatusReg1(void)
{
    unsigned char val;
    SPIFlash_Set_CS(0);
    SPISendByte(0x05);
    val = SPIRecvByte();
    SPIFlash_Set_CS(1);
    return val;
}

static unsigned char SPIFlashReadStatusReg2(void)
{
    unsigned char val;
    SPIFlash_Set_CS(0);
    SPISendByte(0x35);
```

```
        val = SPIRecvByte();
        SPIFlash_Set_CS(1);
        return val;
    }
```

写状态寄存器则是先发出 0x01 命令，再依次发送状态寄存器 1、状态寄存器 2：




```
    static void SPIFlashWriteStatusReg(unsigned char reg1, unsigned char reg2)
    {
        SPIFlashWriteEnable(1);

        SPIFlash_Set_CS(0);
        SPISendByte(0x01);
        SPISendByte(reg1);
        SPISendByte(reg2);
        SPIFlash_Set_CS(1);

        SPIFlashWaitWhenBusy();
    }
```

写状态寄存器还需要去保护，默认的是发出`SPIFlashWriteEnable()`后，即可写状态寄存器，但为了确保万无一失，还是手动在将 SRP1 和 SRP2 设置为 0，即将状态寄存器 1 的最高位清零和状态寄存器最低位清零：

SRP1	SRP0	/WP	Status Register	Description
0	0	X	Software Protection	/WP pin has no control. The Status register can be written to after a Write Enable instruction, WEL=1. [Factory Default]
0	1	0	Hardware Protected	When /WP pin is low the Status Register locked and can not be written to.
0	1	1	Hardware Unprotected	When /WP pin is high the Status register is unlocked and can be written to after a Write Enable instruction, WEL=1.
1	0	X	Power Supply Lock-Down	Status Register is protected and can not be written to again until the next power-down, power-up cycle.[1]
1	1	X	One Time Program[2]	Status Register is permanently protected and can not be written to.

Notes:
1. When SRP1, SRP0 = (1, 0), a power-down, power-up cycle will change SRP1, SRP0 to (0, 0) state.
2. This feature is available upon special order. Please contact Winbond for details.

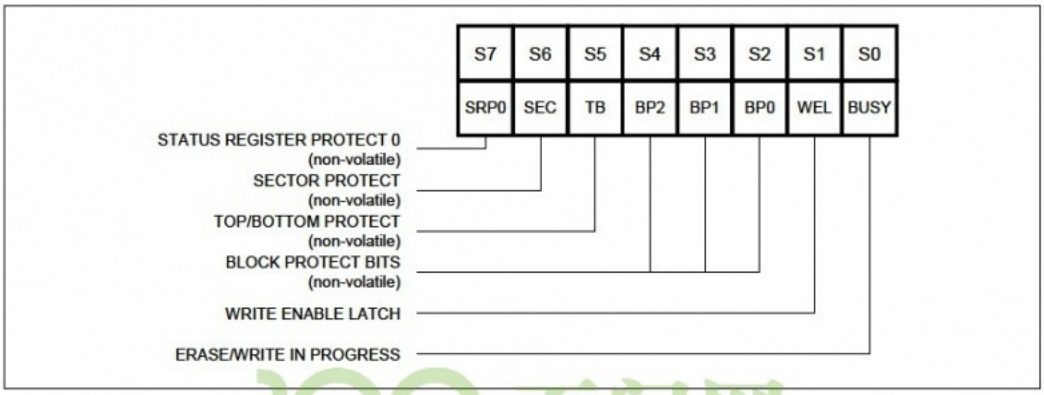


Figure 3a. Status Register-1

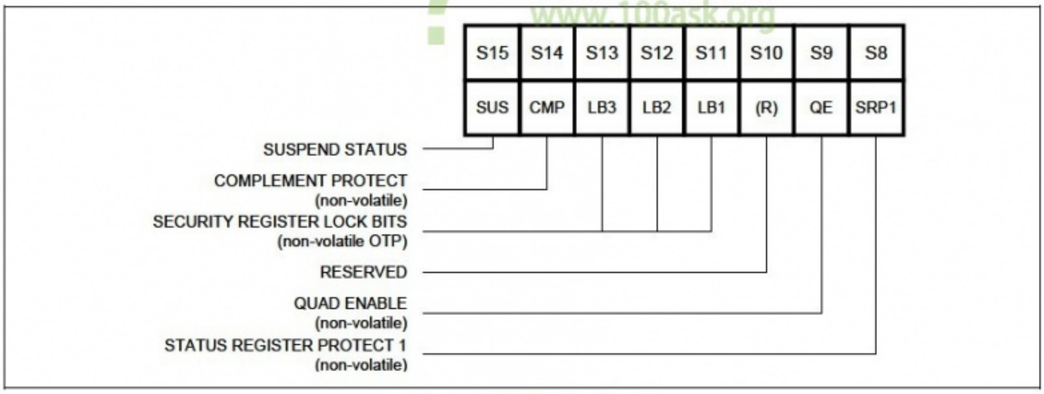


Figure 3b. Status Register-2

```
static void SPIFlashClearProtectForStatusReg(void)
{
    unsigned char reg1, reg2;

    reg1 = SPIFlashReadStatusReg1();
    reg2 = SPIFlashReadStatusReg2();
```

```
        reg1 &= ~(1<<7);
        reg2 &= ~(1<<0);

        SPIFlashWriteStatusReg(reg1, reg2);
    }
```

Flash 有两种保护机制，一个是保护状态寄存器，一种是保护存储数据，现在再来清除数据保护。 需要将 CMP 设置为 0 的同时，将 BP0、BP1、BP2 都设置为 0：

7.1.11 Status Register Memory Protection (CMP = 0)

STATUS REGISTER[1]					W25Q16DV (16M-BIT) MEMORY PROTECTION[3]			
SEC	TB	BP2	BP1	BP0	PROTECTED BLOCK(S)	PROTECTED ADDRESSES	PROTECTED DENSITY	PROTECTED PORTION[2]
X	X	0	0	0	NONE	NONE	NONE	NONE

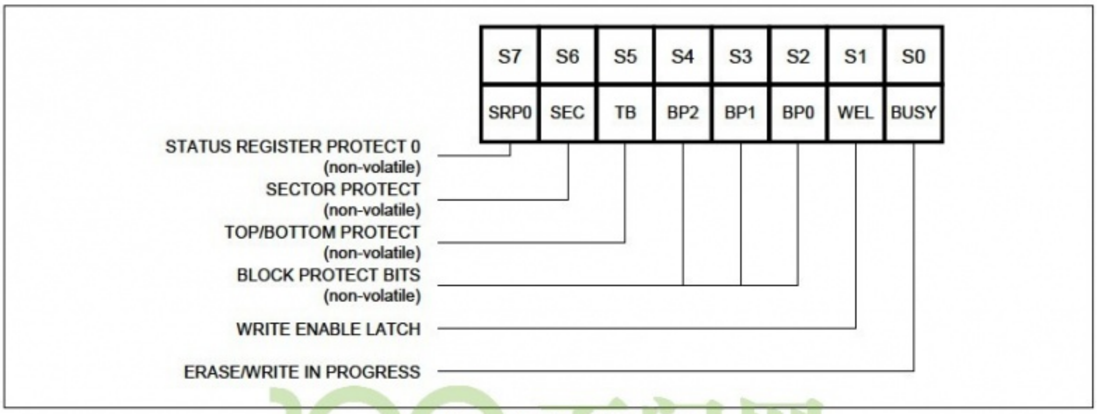


Figure 3a. Status Register-1

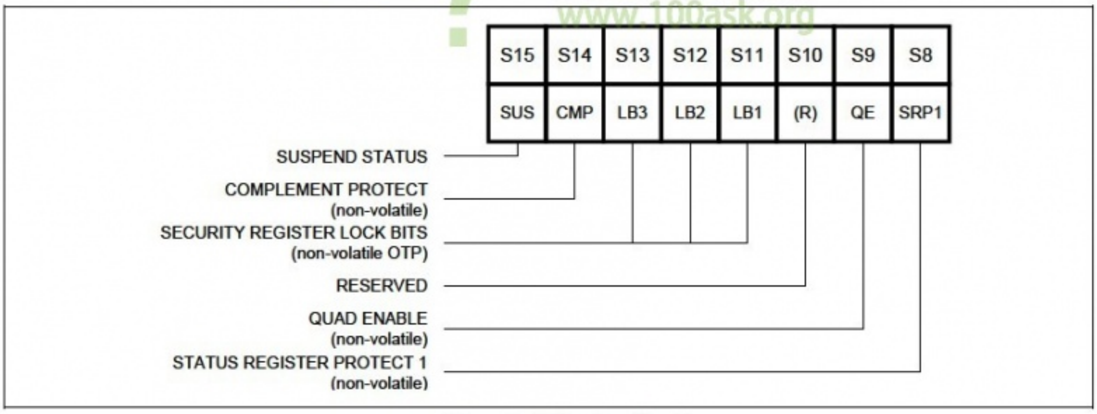


Figure 3b. Status Register-2

```
static void SPIFlashClearProtectForData(void)
{
    /* cmp=0,bp2,1,0=0b000 */
```

```
    unsigned char reg1, reg2;

    reg1 = SPIFlashReadStatusReg1();
    reg2 = SPIFlashReadStatusReg2();

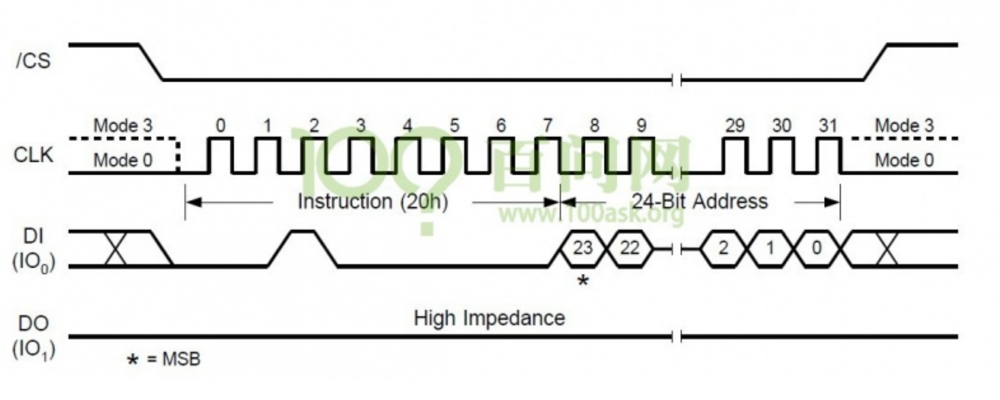
    reg1 &= ~(7<<2);
    reg2 &= ~(1<<6);

    SPIFlashWriteStatusReg(reg1, reg2);
}
```

将两个清除写保护都放在一起，作为一个 SPI Flash 初始化函数：

```
void SPIFlashInit(void)
{
    SPIFlashClearProtectForStatusReg();
    SPIFlashClearProtectForData();
}
```

再来实现擦除，擦除命令需要先发一个 0x20 的命令，再发出 24 位的想擦除位置的地
址：




```
/* erase 4K */
void SPIFlashEraseSector(unsigned int addr)
{
    SPIFlashWriteEnable(1);

    SPIFlash_Set_CS(0);
    SPISendByte(0x20);
    SPIFlashSendAddr(addr);
    SPIFlash_Set_CS(1);

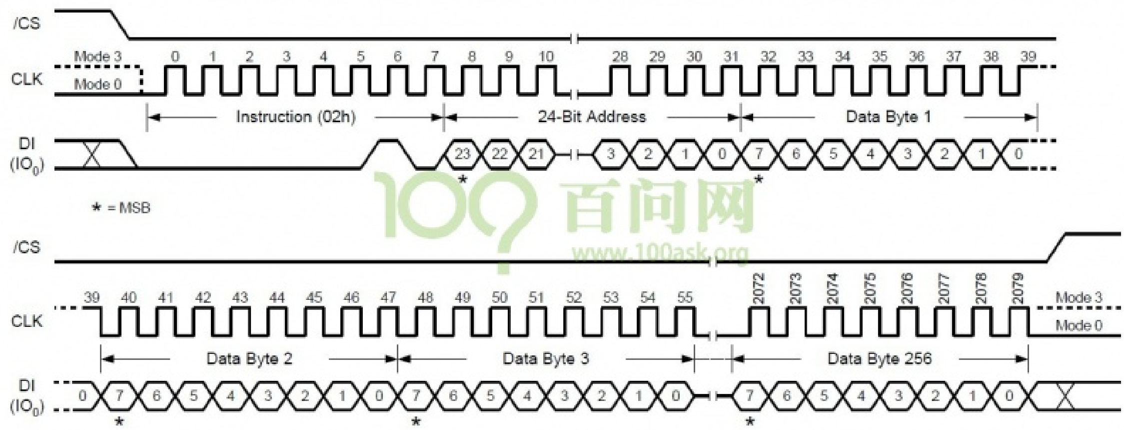
    SPIFlashWaitWhenBusy();
```

```
}
```

为了保证擦除成功，需要读取状态寄存器 1 的的第 1 位：

```
static void SPIFlashWaitWhenBusy(void)
{
    while (SPIFlashReadStatusReg1() & 1);
}
```

然后是烧写函数，先发命令 0x02，再发出 24 位地址，最后再逐个发送数据：




```
/* program */
void SPIFlashProgram(unsigned int addr, unsigned char
*buf, int len)
{
    int i;

    SPIFlashWriteEnable(1);

    SPIFlash_Set_CS(0);
    SPISendByte(0x02);
    SPIFlashSendAddr(addr);

    for (i = 0; i < len; i++)
        SPISendByte(buf[i]);

    SPIFlash_Set_CS(1);

    SPIFlashWaitWhenBusy();

}
```

同前面的擦除操作一样，烧写操作也不是一定是实时的，需要读取状态标志位来判断是否完成。

读函数也是类似的操作，先发命令 0x03，再发出 24 位地址，再逐个读取数据：




```
void SPIFlashRead(unsigned int addr, unsigned char *buf, int len)
{
    int i;

    SPIFlash_Set_CS(0);
    SPISendByte(0x03);
    SPIFlashSendAddr(addr);
    for (i = 0; i < len; i++)
        buf[i] = SPIRecvByte();

    SPIFlash_Set_CS(1);
}
```

至此，基本的 Flash 读写功能已经完成，在主函数调用擦除函数擦除 4096 这个扇区的数据，再往 4096 这个地方写入字符串，再从该地址读取出来，在串口和 OLED 打印出来：

```
SPIFlashEraseSector(4096);
SPIFlashProgram(4096, "100ask", 7);
SPIFlashRead(4096, str, 7);
printf("SPI Flash read from 4096: %s\n\r", str);
OLEDPrint(4,0,str);
```

第 005 节_在 OLED 上显示 ADC 的值

这节我们在 OLED 显示 ADC 电压值，通过调节可调电阻，让 ADC 的值在屏幕上不断变化。 在 JZ2440 的主光盘的 hardware 里面有一个 adc_ts 触摸屏的程序，把里面的`adc_ts.c`和`adc_ts.h`提取出来放在本节视频待写的代码里面。 主函数调用的是`Test_Adc.c`进行测试 adc，因此在里面加上打印和 OLED 显示函数。

```c
/*
 * 测试ADC
 * 通过A/D转换，测量可变电阻器的电压值
 */
void Test_Adc(void)
{
    float vol0, vol1;
    int t0, t1;
    char buf[100];

    printf("Measuring the voltage of AIN0 and AIN1, press any key to exit\n\r");
    while (!awaitkey(0))    // 串口无输入，则不断测试
    {
        vol0 = ((float)ReadAdc(0)*3.3)/1024.0;  // 计算电压值
        vol1 = ((float)ReadAdc(1)*3.3)/1024.0;  // 计算电压值
        t0   = (vol0 - (int)vol0) * 1000;   // 计算小数部分，本代码中的printf无法打印浮点数
        t1   = (vol1 - (int)vol1) * 1000;   // 计算小数部分， 本代码中的printf无法打印浮点数
        printf("AIN0 = %d.%-3dV    AIN1 = %d.%-3dV\r", (int)vol0, t0, (int)vol1, t1);
        sprintf(buf,"ADC: %d.%-3d, %d.%-3d", (int)vol0, t0, (int)vol1, t1);
        OLEDPrint(6, 0, buf);
    }
    printf("\n");
}
```

这里调用了一个`awaitkey()`函数，需要再复制adc_ts触摸屏的程序里`serial.c`的该函数到本工程里面。

```c
/*
 * 接收字符，若有数据直接返回，否则等待规定的时间
 * 输入参数：
 *     timeout: 等待的最大循环次数，0 表示不等待
 * 返回值:
 *    0      : 无数据，超时退出
 *    其他值：串口接收到的数据
 */
unsigned char awaitkey(unsigned long timeout)
{
    while (!(UTRSTAT0 & RXD0READY))
    {
        if (timeout > 0)
            timeout--;
        else
            return 0;   // 超时，返回 0
    }

    return URXH0;       // 返回接收到的串口数据
}
```

修改 Makefile，加入`adc_ts.o`，编译，报错，涉及除法操作，需要加入数学库：

LDFLASG := -L $(shell dirname `$(CC) $(CFLAGS) -print-libgcc-file-name`) -lgcc

现在重新编译即可通过。

现在将 IIC 的的结果也在 OLED 上显示出来，在主函数添加如下代码：

```c
i2c_init();

at24cxx_write(0, 0x55);
data = at24cxx_read(0);

OLEDClearPage(2);
```

```
    OLEDClearPage(3);

    if (data == 0x55)
        OLEDPrint(2,0,"I2C OK!");
    else
        OLEDPrint(2,0,"I2C Err!");
```

先初始化 iic，在 0 地址写入 0x55，然后再读取出来，判断是否与写入的一样，一样则打印 OK，否则打印 Err。

为了防止 OLED 出现之前显示的数据残留，需要再写一个清除 Page 的函数：

```
void OLEDClearPage(int page)
{
    int i;
    OLEDSetPos(page, 0);
    for (i = 0; i < 128; i++)
        OLEDWriteDat(0);
}
```

第 006 节_使用 SPI 控制器

前面我们都是通过 GPIO 管脚来实现的 SPI 通信，这节我们使用 2440 里面的 GPIO 控制器来实现 SPI 通信。 前面使用 GPIO 发送数据时，是手工的控制时钟线、数据线，我们使用 SPI 控制器的话，只需要 把数据写入寄存器，它就可以帮我自动那些时钟线和数据线，我们继续在上一节的基础上修改，添加一个文件`s3c2440_spi.c`和`s3c2440_spi.h`，同时修改 Makefile，替换`gpio_spi.c`为`s3c2440_spi.o`。

从初始化函数开始，需要管脚初始化和 SPI 控制器初始化：

```
void SPIInit(void)
{
    /* 初始化引脚 */
    SPI_GPIO_Init();

    SPIControllerInit();
}
```

管脚初始化即需要把 SPI 相关的 CLK、MOSI、MISO 配置为对应的功能引脚：

```
static void SPI_GPIO_Init(void)
{
    /* GPF1 OLED_CSn output */
```

```
        GPFCON &= ~(3<<(1*2));
        GPFCON |= (1<<(1*2));
        GPFDAT |= (1<<1);

        /* GPG2 FLASH_CSn output
        * GPG4 OLED_DC   output
        * GPG5 SPIMISO
        * GPG6 SPIMOSI
        * GPG7 SPICLK
        */
        GPGCON &= ~((3<<(2*2)) | (3<<(4*2)) | (3<<(5*2)) | 
(3<<(6*2)) | (3<<(7*2)));
        GPGCON |= ((1<<(2*2)) | (1<<(4*2)) | (3<<(5*2)) | 
(3<<(6*2)) | (3<<(7*2)));
        GPGDAT |= (1<<2);
    }
```

然后是 SPI 控制器的初始化，控制器的初始化可以参考芯片手册介绍的编程步骤：

PROGRAMMING PROCEDURE

When a byte data is written into the SPTDATn register, SPI starts to transmit if ENSCK and MSTR of SPCONn register are set. You can use a typical programming procedure to operate an SPI card.

To program the SPI modules, follow these basic steps:

1. Set Baud Rate Prescaler Register (SPPREn).
2. Set SPCONn to configure properly the SPI module.
3. Write data 0xFF to SPTDATn 10 times in order to initialize MMC or SD card.
4. Set a GPIO pin, which acts as nSS, low to activate the MMC or SD card.
5. Tx data ¡ Check the status of Transfer Ready flag (REDY=1), and then write data to SPTDATn.
6. Rx data(1): SPCONn's TAGD bit disable = normal mode
7. ¡ write 0xFF to SPTDATn, then confirm REDY to set, and then read data from Read Buffer.
8. Rx data(2): SPCONn's TAGD bit enable = Tx Auto Garbage Data mode
9. ¡ confirm REDY to set, and then read data from Read Buffer (then automatically start to transfer).
10. Set a GPIO pin, which acts as nSS, high to deactivate the MMC or SD card.

首先是设置波特率，要根据外设所能接受的范围来设置，比如查阅 OLED 的芯片手册得知其时钟最小值为 100ns，即最小为 10MHz；Flash 时钟支持最大 104MHz，为了代码简单，就直接取 10MHz，根据等式推出寄存器值：

```
Baud rate = PCLK / 2 / (Prescaler value + 1)
10 = 50 / 2 / (Prescaler value + 1)
Prescaler value = 1.5 = 2
```

实际的波特率为：50/2/3=8.3MHz

根据参考流程，接下来设置 SPI 控制寄存器：

PROGRAMMING PROCEDURE

When a byte data is written into the SPTDATn register, SPI starts to transmit if ENSCK and MSTR of SPCONn register are set. You can use a typical programming procedure to operate an SPI card.

To program the SPI modules, follow these basic steps:

1. Set Baud Rate Prescaler Register (SPPREn).
2. Set SPCONn to configure properly the SPI module.
3. Write data 0xFF to SPTDATn 10 times in order to initialize MMC or SD card.
4. Set a GPIO pin, which acts as nSS, low to activate the MMC or SD card.
5. Tx data ¡ Check the status of Transfer Ready flag (REDY=1), and then write data to SPTDATn.
6. Rx data(1): SPCONn's TAGD bit disable = normal mode
7. ¡ write 0xFF to SPTDATn, then confirm REDY to set, and then read data from Read Buffer.
8. Rx data(2): SPCONn's TAGD bit enable = Tx Auto Garbage Data mode
9. ¡ confirm REDY to set, and then read data from Read Buffer (then automatically start to transfer).
10. Set a GPIO pin, which acts as nSS, high to deactivate the MMC or SD card.

[6:5]设置为查询模式: 00 polling mode
[4]设置时钟使能: 1 = enable
[3]设置为主机模式: 1 = master
[2]设置无数据时时钟为低电平: 0
[1]设置工作模式为模式 A: 0 = format A
[0]设置发送数据时无需读取数据: 0 = normal mode

```
static void SPIControllerInit(void)
{
    /* OLED  : 100ns, 10MHz
     * FLASH : 104MHz

     * 取 10MHz

     * 10 = 50 / 2 / (Prescaler value + 1)
     * Prescaler value = 1.5 = 2
     * Baud rate = 50/2/3=8.3MHz
     */
    SPPRE0 = 2;
    SPPRE1 = 2;

    /* [6:5] : 00, polling mode
     * [4]   : 1 = enable
     * [3]   : 1 = master
     * [2]   : 0
     * [1]   : 0 = format A
     * [0]   : 0 = normal mode
     */
    SPCON0 = (1<<4) | (1<<3);
    SPCON1 = (1<<4) | (1<<3);
```

```
}
```

发送数据时，先检查状态寄存器，判断发送/接收数据是否准备好了，准备好后就把数据放在寄存器`SPTDAT1`里，SPI 控制器就自己控制时序把数据自动发送出去了。

```
void SPISendByte(unsigned char val)
{
    while (!(SPSTA1 & 1));
    SPTDAT1 = val;
}
```

接收数据时，先写 0xFF 到寄存器`SPTDAT1`，再检查状态寄存器，判断发送/接收数据是否准备好了，准备好后就读取寄存器`SPTDAT1`，读取出来的就是接收到的数据。

```
unsigned char SPIRecvByte(void)
{
    SPTDAT1 = 0xff;
    while (!(SPSTA1 & 1));
    return SPRDAT1;
}
```

第 007 节_移植到 MINI2440_TQ2440

前面在 JZ2440 上操作了 SPI Flash 和 OLED，这节视频是件前面的代码移植到 MINI2440 和 TQ2440 上，如果你使用的是 JZ2440，本节视频就不用看了。

MINI2440 和 TQ2440 上的 SPI 管脚是完全一样的，因此只需移植一个，两者就通用了，先移植 GPIO 模式版本的，复制前面 04th_spi_i2c_adc_jz2440_ok_020_005 里的代码，复制后的新的命名为 06th_spi_i2c_adc_mini2440_tq2440_gpio_020_007 。

修改 gpio_spi.c ，里面的管脚几乎都变化了，因此需要改 SPI_GPIO_Init() ：

```
static void SPI_GPIO_Init(void)
{
    /* GPG1 OLED_CSn output
    * GPG10 FLASH_CSn output
    */
    GPGCON &= ~((3<<(1*2)) | (3<<(10*2)));
    GPGCON |= (1<<(1*2)) | (1<<(10*2));
    GPGDAT |= (1<<1) | (1<<10);

    /*
```

```
    * GPF3    OLED_DC      output
    * GPE11 SPIMISO      input
    * GPE12 SPIMOSI      output
    * GPE13 SPICLK       output
    */
    GPFCON &= ~(3<<(3*2));
    GPFCON |= (1<<(3*2));

    GPECON &= ~((3<<(11*2)) | (3<<(12*2)) | (3<<(13*2)));
    GPECON |= ((1<<(12*2)) | (1<<(13*2)));
}
```

CLK 引脚也变了，修改如下：

```
static void SPI_Set_CLK(char val)
{
    if (val)
        GPEDAT |= (1<<13);
    else
        GPEDAT &= ~(1<<13);
}
```

SPI 的 MOSI 和 MISO 也要变化如下：

```
static void SPI_Set_DO(char val)
{
    if (val)
        GPEDAT |= (1<<12);
    else
        GPEDAT &= ~(1<<12);
}

static char SPI_Get_DI(void)
{
    if (GPEDAT & (1<<11))
        return 1;
    else
        return 0;
}
```

对于 SPI Flash 需要修改其片选引脚，修改 spi_flash.c 里面的片选函数如下:

```
static void SPIFlash_Set_CS(char val)
{
    if (val)
        GPGDAT |= (1<<10);
```

```
    else
        GPGDAT &= ~(1<<10);
}
```

重新编译烧写，测试正常。 再移植 SPI 控制器版本的，复制前面 05th_spi_i2c_adc_jz2440_spi_controller_020_006 里的代码，复制后的新的命名为 07th_spi_i2c_adc_mini2440_tq2440_spi_controller_020_007 。 同样的首先修改 GPIO 初始化，修改为配套引脚：

```
static void SPI_GPIO_Init(void)
{
    /* GPG1 OLED_CSn output
    * GPG10 FLASH_CSn output
    */
    GPGCON &= ~((3<<(1*2)) | (3<<(10*2)));
    GPGCON |= (1<<(1*2)) | (1<<(10*2));
    GPGDAT |= (1<<1) | (1<<10);

    /*
    * GPF3    OLED_DC      output
    * GPE11 SPIMISO
    * GPE12 SPIMOSI
    * GPE13 SPICLK
    */
    GPFCON &= ~(3<<(3*2));
    GPFCON |= (1<<(3*2));

    GPECON &= ~((3<<(11*2)) | (3<<(12*2)) | (3<<(13*2)));
    GPECON |= ((2<<(11*2)) | (2<<(12*2)) | (2<<(13*2)));
}
```

SPI Flash 使用的是 SPI0，因此将 SPTDAT1 改为 *SPTDAT1* :

```
void SPISendByte(unsigned char val)
{
    while (!(SPSTA0 & 1));
    SPTDAT0 = val;
}

unsigned char SPIRecvByte(void)
{
    SPTDAT0 = 0xff;
    while (!(SPSTA0 & 1));
    return SPRDAT0;
}
```

修改 SPI Flash 的片选引脚：

```
static void SPIFlash_Set_CS(char val)
{
	if (val)
		GPGDAT |= (1<<10);
	else
		GPGDAT &= ~(1<<10);
}
```
最后是 OLED 的片选和数据/命令控制引脚：
```
static void OLED_Set_DC(char val)
{
	if (val)
		GPFDAT |= (1<<3);
	else
		GPFDAT &= ~(1<<3);
}

static void OLED_Set_CS(char val)
{
	if (val)
		GPGDAT |= (1<<1);
	else
		GPGDAT &= ~(1<<1);
}
```
重新编译、烧写，测试。